Advanced Java 2
Platform
HOW TO PROGRAM
Deitel™ Books, Cyber Classrooms, Complete Training Courses and Web-Based Training published by Prentice Hall

How to Program Series
Advanced Java™ 2 Platform How to Program
C How to Program, 3/E
C++ How to Program, 3/E
C# How to Program
e-Business and e-Commerce How to Program
Internet and World Wide Web How to Program, 2/E
Java™ How to Program, 4/E
Perl How to Program
Visual Basic® 6 How to Program
Visual Basic® .NET How to Program
Visual C++® .NET How to Program
Wireless Internet & Mobile Business How to Program
XML How to Program

Multimedia Cyber Classroom and Web-Based Training Series
(for information regarding Deitel™ Web-based training visit www.ptgtraining.com)
Advanced Java™ 2 Platform Multimedia Cyber Classroom
C++ Multimedia Cyber Classroom, 3/E
C# Multimedia Cyber Classroom, 3/E
e-Business and e-Commerce Multimedia Cyber Classroom
Internet and World Wide Web Multimedia Cyber Classroom, 2/E
Java™ 2 Multimedia Cyber Classroom, 4/E
Perl Multimedia Cyber Classroom
Visual Basic® 6 Multimedia Cyber Classroom
Visual Basic® .NET Multimedia Cyber Classroom
Visual C++® .NET Multimedia Cyber Classroom

The Complete Training Course Series
The Complete Advanced Java™ 2 Platform Training Course
The Complete C++ Training Course, 3/E
The Complete C# Training Course, 3/E
The Complete e-Business and e-Commerce Programming Training Course
The Complete Internet and World Wide Web Programming Training Course
The Complete Java™ 2 Training Course, 3/E
The Complete Perl Training Course
The Complete Visual Basic® 6 Training Course
The Complete Visual Basic® .NET Training Course
The Complete Visual C++® .NET Training Course
The Complete Wireless Internet & Mobile Business Programming Training Course
The Complete XML Training Course

.NET Series
C# How to Program
Visual Basic® .NET How to Program
Visual C++® .NET How to Program

Visual Studio® Series
Getting Started with Microsoft® Visual C++™ 6 with an Introduction to MFC
Visual Basic® 6 How to Program
C# How to Program
Visual Basic® .NET How to Program
Visual C++® .NET How to Program

For Managers Series
e-Business and e-Commerce for Managers

Coming Soon
e-books and e-whitepapers

To communicate with the authors, send email to:
delite@deitel.com

For information on corporate on-site seminars and public seminars offered by Deitel & Associates, Inc. worldwide, visit:
www.deitel.com

For continuing updates on Prentice Hall and Deitel & Associates, Inc. publications visit the Prentice Hall Web site
www.prenhall.com/deitel
Advanced Java 2
Platform
HOW TO PROGRAM

H. M. Deitel
Deitel & Associates, Inc.

P. J. Deitel
Deitel & Associates, Inc.

S. E. Santry
Deitel & Associates, Inc.
In loving memory of our Uncle and Granduncle Joseph Deitel:

“His pleasure was giving.”

*Harvey and Paul Deitel*

For my brother Tim, who, by his example, always has challenged me to excel.

*Sean*
Trademarks

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Prentice Hall is independent of Sun Microsystems, Inc.

Copyright © 2000 Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399 U.S.A. All rights reserved.

Netscape Communicator browser window© 1999 Netscape Communications Corporation. Used with permission. Netscape Communications has not authorized, sponsored, endorsed, or approved this publication and is not responsible for its content.

Openwave, the Openwave logo, and UP.SDK are trademarks of Openwave Systems Inc. All rights reserved."

Palm OS, Palm Computing, HandFAX, HandSTAMP, HandWEB, Graffiti, HotSync, iMEssenger, MultiMail, Palm.Net, PalmConnect, PalmGlove, PalmModem, PalmPoint, PalmPrint, PalmSource, and the Palm Platform Compatible Logo are registered trademarks of Palm, Inc. Palm, the Palm logo, MyPalm, PalmGear, PalmPix, PalmPower, AnyDay, EventClub, HandMAIL, the HotSync Logo, PalmGlove, Palm Powered, the Palm trade dress, Smartcode, Simply Palm, We Sync and Wireless Refresh are trademarks of Palm, Inc.
Contents

Preface

7 Introduction

1.1 Introduction
1.2 Architecture of the Book
1.2.1 Advanced GUI, Graphics and JavaBeans
1.2.2 Distributed Systems
1.2.3 Web Services
1.2.4 Enterprise Java
1.2.5 Enterprise Case Study
1.2.6 XML
1.3 Tour of the Book
1.4 Running Example Code
1.5 Design Patterns
1.5.1 History of Object-Oriented Design Patterns
1.5.2 Design Patterns Discussion
1.5.3 Concurrency Patterns
1.5.4 Architectural Patterns
1.5.5 Further Study on Design Patterns

2 Advanced Swing Graphical User Interface Components

2.1 Introduction
2.2 WebBrowser Using JEditorPane and JToolBar
2.2.1 Swing Text Components and HTML Rendering
2.2.2 Swing Toolbars
2.3 Swing Actions
2.4 JSplitPane and JTabbedPane
2.5 Multiple-Document Interfaces 52
2.6 Drag and Drop 56
2.7 Internationalization 62
2.8 Accessibility 71
2.9 Internet and World Wide Web Resources 78

3 Model-View-Controller 85
3.1 Introduction 86
3.2 Model-View-Controller Architecture 86
3.3 Observable Class and Observer Interface 88
3.4 JList 107
3.5 JTable 111
3.6 JTree 115
3.6.1 Using DefaultTreeModel 117
3.6.2 Custom TreeModel Implementation 123

4 Graphics Programming with Java 2D and Java 3D 135
4.1 Introduction 136
4.2 Coordinates, Graphics Contexts and Graphics Objects 136
4.3 Java 2D API 138
4.3.1 Java 2D Shapes 140
4.3.2 Java 2D Image Processing 146
4.4 Java 3D API 160
4.4.1 Obtaining and Installing the Java 3D API 161
4.4.2 Java 3D Scenes 161
4.4.3 A Java 3D Example 163
4.5 A Java 3D Case Study: A 3D Game with Custom Behaviors 179

5 Case Study: Java 2D GUI Application with Design Patterns 219
5.1 Introduction 220
5.2 Application Overview 220
5.3 MyShape Class Hierarchy 221
5.4 Deitel DrawingModel 242
5.5 Deitel Drawing Views 254
5.6 Deitel Drawing Controller Logic 260
5.6.1 MyShapeControllers for Processing User Input 260
5.6.2 MyShapeControllers and Factory Method Design Pattern 272
5.6.3 Drag-and-Drop Controller 276
5.7 DrawingInternalFrame Component 287
5.8 ZoomDialog, Action and Icon Components 304
5.9 DeitelDrawing Application 309

6 JavaBeans Component Model 321
6.1 Introduction 322
6.2 Using Beans in Forte for Java Community Edition 323
Contents

6.3 Preparing a Class to be a JavaBean 337
6.4 Creating a JavaBean: Java Archive Files 340
6.5 JavaBean Properties 345
6.6 Bound Properties 347
6.7 Indexed Properties and Custom Events 355
6.8 Customizing JavaBeans for Builder Tools 364
   6.8.1 PropertyEditors 371
   6.8.2 Customizers 375
6.9 Internet and World Wide Web Resources 379

7 Security 386
7.1 Introduction 387
7.2 Ancient Ciphers to Modern Cryptosystems 388
7.3 Secret-Key Cryptography 389
7.4 Public-Key Cryptography 390
7.5 Cryptanalysis 393
7.6 Key Agreement Protocols 393
7.7 Key Management 394
7.8 Java Cryptography Extension (JCE) 395
   7.8.1 Password-Based Encoding with JCE 395
   7.8.2 Decorator Design Pattern 405
7.9 Digital Signatures 406
7.10 Public-Key Infrastructure, Certificates and Certification Authorities 407
   7.10.1 Java Keystores and keytool 409
7.11 Java Policy Files 410
7.12 Digital Signatures for Java Code 413
7.13 Authentication 417
   7.13.1 Kerberos 417
   7.13.2 Single Sign-On 417
   7.13.3 Java Authentication and Authorization Service (JAAS) 418
7.14 Secure Sockets Layer (SSL) 423
   7.14.1 Java Secure Socket Extension (JSSE) 424
7.15 Java Language Security and Secure Coding 429
7.16 Internet and World Wide Web Resources 430

8 Java Database Connectivity (JDBC) 444
8.1 Introduction 445
8.2 Relational-Database Model 446
8.3 Relational Database Overview: The books Database 447
8.4 Structured Query Language (SQL) 452
   8.4.1 Basic SELECT Query 453
   8.4.2 WHERE Clause 454
   8.4.3 ORDER BY Clause 456
   8.4.4 Merging Data from Multiple Tables: Joining 459
   8.4.5 INSERT INTO Statement 460
   8.4.6 UPDATE Statement 461
   8.4.7 DELETE FROM Statement 462
8.5 Creating Database books in Cloudscape 463
8.6 Manipulating Databases with JDBC 464
  8.6.1 Connecting to and Querying a JDBC Data Source 464
  8.6.2 Querying the books Database 470
8.7 Case Study: Address-Book Application 479
  8.7.1 PreparedStatements 480
  8.7.2 Transaction Processing 482
  8.7.3 Address-Book Application 482
8.8 Stored Procedures 515
8.9 Batch Processing 515
8.10 Processing Multiple ResultSet objects or Update Counts 517
8.11 Updatable ResultSets 518
8.12 JDBC 2.0 Optional Package javax.sql 519
  8.12.1 DataSource 519
  8.12.2 Connection Pooling 519
  8.12.3 RowSets 520
8.13 Internet and World Wide Web Resources 520

9 Servlets 530
9.1 Introduction 531
9.2 Servlet Overview and Architecture 533
  9.2.1 Interface Servlet and the Servlet Life Cycle 534
  9.2.2 HttpServlet Class 536
  9.2.3 HttpServletRequest Interface 537
  9.2.4 HttpServletResponse Interface 538
9.3 Handling HTTP get Requests 539
  9.3.1 Setting Up the Apache Tomcat Server 543
  9.3.2 Deploying a Web Application 545
9.4 Handling HTTP get Requests Containing Data 549
9.5 Handling HTTP post Requests 552
9.6 Redirecting Requests to Other Resources 556
9.7 Session Tracking 559
  9.7.1 Cookies 560
  9.7.2 Session Tracking with HttpSession 569
9.8 Multi-Tier Applications: Using JDBC from a Servlet 577
9.9 HttpUtils Class 584
9.10 Internet and World Wide Web Resources 585

10 JavaServer Pages (JSP) 593
10.1 Introduction 594
10.2 JavaServer Pages Overview 595
10.3 A First JavaServer Page Example 596
10.4 Implicit Objects 598
10.5 Scripting 599
  10.5.1 Scripting Components 600
  10.5.2 Scripting Example 601
10.6 Standard Actions 604
10.6.1  `<jsp:include>` Action 605
10.6.2  `<jsp:forward>` Action 610
10.6.3  `<jsp:plugin>` Action 613
10.6.4  `<jsp:useBean>` Action 617
10.7  Directives 634
   10.7.1  `page` Directive 634
   10.7.2  `include` Directive 636
10.8  Custom Tag Libraries 638
   10.8.1  Simple Custom Tag 639
   10.8.2  Custom Tag with Attributes 643
   10.8.3  Evaluating the Body of a Custom Tag 647
10.9  Internet and World Wide Web Resources 653

11  Case Study: Servlet and JSP Bookstore 660
  11.1  Introduction 661
  11.2  Bookstore Architecture 662
  11.3  Entering the Bookstore 664
  11.4  Obtaining the Book List from the Database 667
  11.5  Viewing a Book’s Details 676
  11.6  Adding an Item to the Shopping Cart 683
  11.7  Viewing the Shopping Cart 686
  11.8  Checking Out 689
  11.9  Processing the Order 693
  11.10  Deploying the Bookstore Application in J2EE 1.2.1 694
     11.10.1  Configuring the books Data Source 695
     11.10.2  Launching the Cloudscape Database and J2EE Servers 695
     11.10.3  Launching the J2EE Application Deployment Tool 696
     11.10.4  Creating the Bookstore Application 697
     11.10.5  Creating BookServlet and AddToCartServlet Web Components 698
     11.10.6  Adding Non-Servlet Components to the Application 704
     11.10.7  Specifying the Web Context, Resource References, JNDI 706
       Names and Welcome Files
     11.10.8  Deploying and Executing the Application 709

12  Java-Based Wireless Applications Development and J2ME 716
  12.1  Introduction 717
  12.2  `WelcomeServlet` Overview 720
  12.3  `TipTestServlet` Overview 726
     12.3.1  Internet Explorer Request 739
     12.3.2  WAP Request 746
     12.3.3  Pixo i-mode Request 751
     12.3.4  J2ME Client Request 755
  12.4  Java 2 Micro Edition 757
     12.4.1  Connected Limited Device Configuration (CLDC) 758
     12.4.2  Mobile Information Device Profile (MIDP) 759
## 16 Messaging with JMS

16.1 Introduction  
16.2 Installation and Configuration of J2EE 1.3  
16.3 Point-To-Point Messaging  
  16.3.1 Voter Application: Overview  
  16.3.2 Voter Application: Sender Side  
  16.3.3 Voter Application: Receiver Side  
  16.3.4 Voter Application: Configuring and Running  
16.4 Publish/Subscribe Messaging  
  16.4.1 Weather Application: Overview  
  16.4.2 Weather Application: Publisher Side  
  16.4.3 Weather Application: Subscriber Side  
  16.4.4 Weather Application: Configuring and Running  
16.5 Message-Driven Enterprise JavaBeans  
  16.5.1 Voter Application: Overview  
  16.5.2 Voter Application: Receiver Side  
  16.5.3 Voter Application: Configuring and Running

## 17 Enterprise Java Case Study: Architectural Overview

17.1 Introduction  
17.2 Deitel Bookstore  
17.3 System Architecture  
17.4 Enterprise JavaBeans  
  17.4.1 Entity EJBs  
  17.4.2 Stateful Session EJBs  
17.5 Servlet Controller Logic  
17.6 XSLT Presentation Logic

## 18 Enterprise Java Case Study: Presentation and Controller Logic

18.1 Introduction  
18.2 `XMLServlet` Base Class  
18.3 Shopping Cart Servlets  
  18.3.1 `AddToCartServlet`  
  18.3.2 `ViewCartServlet`  
  18.3.3 `RemoveFromCartServlet`  
  18.3.4 `UpdateCartServlet`  
  18.3.5 `CheckoutServlet`  
18.4 Product Catalog Servlets  
  18.4.1 `GetAllProductsServlet`  
  18.4.2 `GetProductServlet`  
  18.4.3 `ProductSearchServlet`  
18.5 Customer Management Servlets  
  18.5.1 `RegisterServlet`  
  18.5.2 `LoginServlet`  
  18.5.3 `ViewOrderHistoryServlet`
18.5.4 ViewOrderServlet
18.5.5 GetPasswordHintServlet

19 Enterprise Java Case Study: Business Logic Part 1 1073
19.1 Introduction 1074
19.2 EJB Architecture 1074
19.3 ShoppingCart Implementation 1075
19.3.1 ShoppingCart Remote Interface 1075
19.3.2 ShoppingCartEJB Implementation 1077
19.3.3 ShoppingCartHome Interface 1084
19.4 Product Implementation 1085
19.4.1 Product Remote Interface 1085
19.4.2 ProductEJB Implementation 1086
19.4.3 ProductHome Interface 1088
19.4.4 ProductModel 1089
19.5 Order Implementation 1094
19.5.1 Order Remote Interface 1095
19.5.2 OrderEJB Implementation 1095
19.5.3 OrderHome Interface 1101
19.5.4 OrderModel 1101
19.6 OrderProduct Implementation 1107
19.6.1 OrderProduct Remote Interface 1107
19.6.2 OrderProductEJB Implementation 1108
19.6.3 OrderProductHome Interface 1110
19.6.4 OrderProductPK Primary-Key Class 1111
19.6.5 OrderProductModel 1113

20 Enterprise Java Case Study: Business Logic Part 2 1117
20.1 Introduction 1118
20.2 Customer Implementation 1118
20.2.1 Customer Remote Interface 1119
20.2.2 CustomerEJB Implementation 1119
20.2.3 CustomerHome Interface 1126
20.2.4 CustomerModel 1127
20.3 Address Implementation 1131
20.3.1 Address Remote Interface 1134
20.3.2 AddressEJB Implementation 1134
20.3.3 AddressHome Interface 1138
20.3.4 AddressModel 1138
20.4 SequenceFactory Implementation 1144
20.4.1 SequenceFactory Remote Interface 1144
20.4.2 SequenceFactoryEJB Implementation 1145
20.4.3 SequenceFactoryHome Interface 1147
20.5 Deitel Bookstore Application Deployment with J2EE 1149
20.5.1 Deploying Deitel Bookstore CMP Entity EJBs 1149
20.5.2 Deploying Deitel Bookstore Servlets 1156
## 21 Application Servers

21.1 Introduction 1162
21.2 J2EE Specification and Benefits 1162
21.3 Commercial Application Servers 1163
   21.3.1 BEA WebLogic 6.0 1163
   21.3.2 iPlanet Application Server 6.0 1164
   21.3.3 IBM WebSphere Advanced Application Server 4.0 1165
   21.3.4 JBoss 2.2.2 Application Server 1165
21.4 Deploying the Deitel Bookstore on BEA WebLogic 1165
21.5 Deploying the Deitel Bookstore on IBM WebSphere 1191
21.6 Internet and World Wide Web Resources 1193

## 22 Jini

22.1 Introduction 1197
22.2 Installing Jini 1198
22.3 Configuring the Jini Runtime Environment 1198
22.4 Starting the Required Services 1199
22.5 Running the Jini LookupBrowser 1203
22.6 Discovery 1204
   22.6.1 Unicast Discovery 1204
   22.6.2 Multicast Discovery 1209
22.7 Jini Service and Client Implementations 1214
   22.7.1 Service Interfaces and Supporting Classes 1214
   22.7.2 Service Proxy and Service Implementations 1217
   22.7.3 Registering the Service with Lookup Services 1220
   22.7.4 Jini Service Client 1223
22.8 Introduction to High-Level Helper Utilities 1232
   22.8.1 Discovery Utilities 1232
   22.8.2 Entry Utilities 1242
   22.8.3 Lease Utilities 1244
   22.8.4 JoinManager Utility 1248
   22.8.5 Service Discovery Utilities 1252
22.9 Internet and World Wide Web Resources 1253

## 23 JavaSpaces

23.1 Introduction 1259
23.2 JavaSpaces Service Properties 1260
23.3 JavaSpaces Service 1260
23.4 Discovering the JavaSpaces Service 1262
23.5 JavaSpace Interface 1264
23.6 Defining an Entry 1265
23.7 Write Operation 1266
23.8 Read and Take Operations 1269
   23.8.1 Read Operation 1269
   23.8.2 Take Operation 1273
23.9 Notify Operation 1276
23.10 Method snapshot 1281
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.11</td>
<td>Updating Entries with Jini Transaction Service</td>
<td>1284</td>
</tr>
<tr>
<td>23.11.1</td>
<td>Defining the User Interface</td>
<td>1285</td>
</tr>
<tr>
<td>23.11.2</td>
<td>Discovering the <strong>TransactionManager</strong> Service</td>
<td>1287</td>
</tr>
<tr>
<td>23.11.3</td>
<td>Updating an <strong>Entry</strong></td>
<td>1289</td>
</tr>
<tr>
<td>23.12</td>
<td>Case Study: Distributed Image Processing</td>
<td>1294</td>
</tr>
<tr>
<td>23.12.1</td>
<td>Defining an Image Processor</td>
<td>1295</td>
</tr>
<tr>
<td>23.12.2</td>
<td>Partitioning an Image into Smaller Pieces</td>
<td>1301</td>
</tr>
<tr>
<td>23.12.3</td>
<td>Compiling and Running the Example</td>
<td>1312</td>
</tr>
<tr>
<td>23.13</td>
<td>Internet and World Wide Web Resources</td>
<td>1314</td>
</tr>
<tr>
<td>24</td>
<td><strong>Java Management Extensions (JMX) (on CD)</strong></td>
<td>1319</td>
</tr>
<tr>
<td>24.1</td>
<td>Introduction</td>
<td>1320</td>
</tr>
<tr>
<td>24.2</td>
<td>Installation</td>
<td>1322</td>
</tr>
<tr>
<td>24.3</td>
<td>Case Study</td>
<td>1322</td>
</tr>
<tr>
<td>24.3.1</td>
<td>Instrument Resources</td>
<td>1322</td>
</tr>
<tr>
<td>24.3.2</td>
<td>Implementation of the JMX Management Agent</td>
<td>1338</td>
</tr>
<tr>
<td>24.3.3</td>
<td>Broadcasting and Receiving Notifications</td>
<td>1342</td>
</tr>
<tr>
<td>24.3.4</td>
<td>Management Application</td>
<td>1346</td>
</tr>
<tr>
<td>24.3.5</td>
<td>Compiling and Running the Example</td>
<td>1357</td>
</tr>
<tr>
<td>24.4</td>
<td>Internet and World Wide Web Resources</td>
<td>1360</td>
</tr>
<tr>
<td>25</td>
<td><strong>Jiro (on CD)</strong></td>
<td>1364</td>
</tr>
<tr>
<td>25.1</td>
<td>Introduction</td>
<td>1365</td>
</tr>
<tr>
<td>25.2</td>
<td>Installation</td>
<td>1366</td>
</tr>
<tr>
<td>25.3</td>
<td>Starting Jiro</td>
<td>1367</td>
</tr>
<tr>
<td>25.4</td>
<td>Dynamic vs. Static Services</td>
<td>1369</td>
</tr>
<tr>
<td>25.5</td>
<td>Dynamic Services</td>
<td>1369</td>
</tr>
<tr>
<td>25.5.1</td>
<td>Dynamic-Service Implementation</td>
<td>1370</td>
</tr>
<tr>
<td>25.6</td>
<td>Static Services</td>
<td>1380</td>
</tr>
<tr>
<td>25.6.1</td>
<td>Locating Static Services with Class <strong>ServiceFinder</strong></td>
<td>1380</td>
</tr>
<tr>
<td>25.6.2</td>
<td>Event Service</td>
<td>1381</td>
</tr>
<tr>
<td>25.6.3</td>
<td>Log Service</td>
<td>1389</td>
</tr>
<tr>
<td>25.6.4</td>
<td>Scheduling Service</td>
<td>1391</td>
</tr>
<tr>
<td>25.7</td>
<td>Dynamic Service Deployment</td>
<td>1392</td>
</tr>
<tr>
<td>25.7.1</td>
<td>Dynamic–Service Usage</td>
<td>1395</td>
</tr>
<tr>
<td>25.8</td>
<td>Management Policies</td>
<td>1409</td>
</tr>
<tr>
<td>25.8.1</td>
<td>Policy–Management Deployment</td>
<td>1420</td>
</tr>
<tr>
<td>25.9</td>
<td>Closing Notes on the Printer Management Solution</td>
<td>1428</td>
</tr>
<tr>
<td>25.10</td>
<td>Internet and World Wide Web Resources</td>
<td>1429</td>
</tr>
<tr>
<td>26</td>
<td><strong>Common Object Request Broker Architecture (CORBA): Part 1 (on CD)</strong></td>
<td>1435</td>
</tr>
<tr>
<td>26.1</td>
<td>Introduction</td>
<td>1436</td>
</tr>
<tr>
<td>26.2</td>
<td>Step-by-Step</td>
<td>1441</td>
</tr>
<tr>
<td>26.3</td>
<td>First Example: <strong>SystemClock</strong></td>
<td>1442</td>
</tr>
<tr>
<td>26.3.1</td>
<td><strong>SystemClock.idl</strong></td>
<td>1443</td>
</tr>
<tr>
<td>26.3.2</td>
<td><strong>SystemClockImpl.java</strong></td>
<td>1444</td>
</tr>
</tbody>
</table>
### 26.3.3 **SystemClockClient.java**

26.3.4 Running the Example

### 26.4 Technical/Architectural Overview

### 26.5 CORBA Basics

### 26.6 Example: **AlarmClock**

26.6.1 **AlarmClock.idl**

26.6.2 **AlarmClockImpl.java**

26.6.3 **AlarmClockClient.java**

### 26.7 Distributed Exceptions

### 26.8 Case Study: Chat

26.8.1 **chat.idl**

26.8.2 **ChatServerImpl.java**

26.8.3 **DeitelMessenger.java**

26.8.4 Running Chat

26.8.5 Issues

### 26.9 Comments and Comparisons

### 26.10 Internet and World Wide Web Resources

### 27 Common Object Request Broker Architecture (CORBA): Part 2 (on CD)

### 27.1 Introduction

### 27.2 Static Invocation Interface (SII), Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)

### 27.3 BOAs, POAs and TIEs

### 27.4 CORBA services

27.4.1 Naming Service

27.4.2 Security Service

27.4.3 Object Transaction Service

27.4.4 Persistent State Service

27.4.5 Event and Notification Services

### 27.5 EJBs and CORBA components

### 27.6 CORBA vs. RMI

27.6.1 When to Use RMI

27.6.2 When to Use CORBA

27.6.3 RMI-IIOP

### 27.7 RMIMessenger Case Study Ported to RMI-IIOP

27.7.1 **ChatServer** RMI-IIOP Implementation

27.7.2 **ChatClient** RMI-IIOP Implementation

27.7.3 Compiling and Running the **ChatServer** and **ChatClient**

### 27.8 Future Directions

### 27.9 Internet and World Wide Web Resources

### 28 Peer-to-Peer Applications and JXTA

### 28.1 Introduction

### 28.2 Client/Server and Peer-to-Peer Applications

### 28.3 Centralized vs. Decentralized Network Applications

### 28.4 Peer Discovery and Searching
28.5 Case Study: Deitel Instant Messenger 1551
28.6 Defining the Service Interface 1553
28.7 Defining the Service implementation 1555
28.8 Registering the Service 1562
28.9 Find Other Peers 1564
28.10 Compiling and Running the Example 1571
28.11 Improving Deitel Instant Messenger 1571
28.12 Deitel Instant Messenger with Multicast Sockets 1572
  28.12.1 Registering the Peer 1572
  28.12.2 Finding Other Peers 1577
28.13 Introduction to JXTA 1588
28.14 Internet and World Wide Web Resources 1590

29 Introduction to Web Services and SOAP 1594
29.1 Introduction 1595
29.2 Simple Object Access Protocol (SOAP) 1596
29.3 SOAP Weather Service 1602
29.4 Internet and World Wide Web Resources 1608

A Creating Markup with XML (on CD) 1611
A.1 Introduction 1612
A.2 Introduction to XML Markup 1612
A.3 Parsers and Well-Formed XML Documents 1615
A.4 Characters 1616
  A.4.1 Characters vs. Markup 1616
  A.4.2 White Space, Entity References and Built-In Entities 1616
A.5 CDATA Sections and Processing Instructions 1618
A.6 XML Namespaces 1620
A.7 Internet and World Wide Web Resources 1623

B Document Type Definition (DTD) (on CD) 1627
B.1 Introduction 1628
B.2 Parsers, Well-Formed and Valid XML Documents 1628
B.3 Document Type Declaration 1629
B.4 Element Type Declarations 1630
  B.4.1 Sequences, Pipe Characters and Occurrence Indicators 1631
  B.4.2 EMPTY, Mixed Content and ANY 1634
B.5 Attribute Declarations 1636
B.6 Attribute Types 1638
  B.6.1 Tokenized Attribute Type (ID, IDREF, ENTITY, NMTOKEN) 1638
  B.6.2 Enumerated Attribute Types 1643
B.7 Conditional Sections 1644
B.8 Whitespace Characters 1645
B.9 Internet and World Wide Web Resources 1647
# Contents

## C  Document Object Model (DOM™) (on CD)  1652
- C.1 Introduction  1653
- C.2 DOM with Java  1654
- C.3 Setup Instructions  1657
- C.4 DOM Components  1657
- C.5 Creating Nodes  1665
- C.6 Traversing the DOM  1668
- C.7 Internet and World Wide Web Resources  1671

## D  XSL: Extensible Stylesheet Language Transformations (XSLT) (on CD)  1676
- D.1 Introduction  1677
- D.2 Applying XSLTs with Java  1677
- D.3 Templates  1679
- D.4 Creating Elements and Attributes  1680
- D.5 Iteration and Sorting  1683
- D.6 Conditional Processing  1687
- D.7 Combining Style Sheets  1690
- D.8 Variables  1695
- D.9 Internet and World Wide Web Resources  1695

## E  Downloading and Installing J2EE 1.2.1 (on CD)  1699
- E.1 Introduction  1699
- E.2 Installation  1699
- E.3 Configuration  1700
  - E.3.1 JDBC Drivers and Data Sources  1700
  - E.3.2 HTTP properties  1700

## F  Java Community ProcessSM (JCP) (on CD)  1701
- F.1 Introduction  1701
- F.2 Participants  1701
  - F.2.1 Program Management Office  1701
  - F.2.2 Executive Committee  1701
  - F.2.3 Experts  1702
  - F.2.4 Members  1702
  - F.2.5 Public Participation  1702
- F.3 Java Community Process  1702
  - F.3.1 Initiation Phase  1702
  - F.3.2 Community Draft Phase  1703
  - F.3.3 Public Draft Phase  1704
  - F.3.4 Final Phase  1704
  - F.3.5 Maintenance Phase  1704

## G  Java Native Interface (JNI) (on CD)  1705
- G.1 Introduction  1706
Illustrations

7 Introduction
1.1 Gang-of-four 23 design patterns. 21
1.2 Gang-of-four design patterns used in *Advanced Java 2 Platform How to Program*. 22

2 Advanced Swing Graphical User Interface Components
2.1 `WebBrowserPane` subclass of `JEditorPane` for viewing Web sites and maintaining URL history. 31
2.2 Toolbars for navigating the Web in Internet Explorer and Mozilla. 34
2.3 `WebToolBar` `JToolBar` subclass for navigating URLs in a `WebBrowserPane`. 35
2.4 `WebBrowser` application for browsing Web sites using `WebBrowserPane` and `WebToolBar`. 38
2.5 `ActionSample` application demonstrating the Command design pattern with Swing `Actions`. 40
2.6 `Action` class static keys for `Action` properties. 45
2.7 `FavoritesWebBrowser` application for displaying two Web pages side-by-side using `JSplitPane`. 45
2.8 Tabbed interface of `Display Properties` dialog box in Windows 2000. 48
2.9 `TabbedPaneWebBrowser` application using `JTabbedPane` to browse multiple Web sites concurrently. 49
2.10 `MDIWebBrowser` application using `JDesktopPane` and `JInternalFrames` to browse multiple Web sites concurrently. 52
2.11 `DnDWebBrowser` application for browsing Web sites that also accepts drag-and-drop operations for viewing HTML pages. 57
2.12 `WebToolBar` that uses `ResourceBundles` for internationalization. 63
2.13 `MyAbstractAction` `AbstractAction` subclass that provides `set` methods for `Action` properties. 66
2.14 **WebBrowser** that uses **ResourceBundles** for internationalization. 67
2.15 **BrowserLauncher** application for selecting a **Locale** and launching an internationalized **WebBrowser**. 68
2.16 Properties file for default **Locale** (US English)—
**StringsAndLabels.properties**. 70
2.17 Properties file for French **Locale**—
**StringsAndLabels_fr_FR.properties**. 71
2.18 **ActionSample2** demonstrates **Accessibility** package. 72
2.19 Actions **sampleAction** and **exitAction** of **ActionSample2**. 77
2.20 **AccessibleDescription** of **sampleButton**. 77
2.21 **AccessibleDescription** of **exitButton**. 78
2.22 **Sample Action** menu item description. 78
2.23 **Exit** menu item description. 78

3 **Model-View-Controller**
3.1 Model-view-controller architecture. 87
3.2 Delegate-model architecture in Java Swing components. 87
3.3 **AccountManager** application MVC architecture. 89
3.4 **Account Observable** class that represents a bank account. 89
3.5 **AbstractAccountView** abstract base class for observing **Account**s. 91
3.6 **AccountTextView** for displaying observed **Account** information in a **JTextField**. 93
3.7 **AccountBarGraphView** for rendering observed **Account** information as a bar graph. 94
3.8 **AssetPieChartView** for rendering multiple observed asset **Account**s as a pie chart. 97
3.9 **AccountController** for obtaining user input to modify **Account** information. 102
3.10 **AccountManager** application for displaying and modifying **Account** information using the model-view-controller architecture. 105
3.11 **JList** and **ListModel** delegate-model architecture. 108
3.12 **PhilosophersJList** application demonstrating **JList** and **DefaultListModel**. 108
3.13 **TableModel** interface methods and descriptions. 111
3.14 **JTable** and **TableModel** delegate-model architecture. 112
3.15 **PhilosophersJTable** application demonstrating **JTable** and **DefaultTableModel**. 112
3.16 **JTree** showing a hierarchy of philosophers. 116
3.17 **PhilosophersJTree** application demonstrating **JTree** and **DefaultTreeModel**. 117
3.18 **FileSystemModel** implementation of interface **TreeModel** to represent a file system. 123
3.19 **FileTreeFrame** application for browsing and editing a file system using **JTree** and **FileSystemModel**. 129
4 Graphics Programming with Java 2D and Java 3D

4.1 Java coordinate system. Units are measured in pixels. 137
4.2 Some Java 2D classes and interfaces. 138
4.3 The seven state attributes of a Java 2D graphics context. 139
4.4 Demonstrating some Java 2D shapes. 140
4.5 Demonstrating Java 2D paths. 144
4.6 Class ImagePanel allows for displaying and filtering BufferedImage. 147
4.7 Java2DImageFilter interface for creating Java 2D image filters. 150
4.8 Classes that implement BufferedImageOp and RasterOp. 151
4.9 InvertFilter inverts colors in a BufferedImage. 152
4.10 SharpenFilter sharpens edges in a BufferedImage. 153
4.11 BlurFilter blurs the colors in a BufferedImage. 153
4.12 ColorFilter changes the colors in a BufferedImage. 154
4.13 Java 2D image-processing application GUI. 156
4.14 Java 3D Group, Leaf and NodeComponent subclasses. 162
4.15 Creating a Java 3D SimpleUniverse with content. 163
4.16 Demonstrating MouseRotate behavior. 170
4.17 Demonstrating MouseTranslate behavior. 171
4.18 Demonstrating MouseZoom behavior. 172
4.19 Demonstrating changing color in Java 3D. 173
4.20 Demonstrating texture mapping in Java 3D. 174
4.21 ControlPanel provides Swing controls for Java3DWorld. 175
4.22 GUI for Java3DWorld and ControlPanel. 179
4.23 Class Java3DWorld1 creates the 3D-game environment. 181
4.24 Implementing collision detection in a Java 3D application. 198
4.25 Behavior that enables the user to navigate a 3D shape. 200
4.26 Keys for navigating the 3D scene in Navigator. 204
4.27 Implementing a position-checking Behavior. 205
4.28 Implementing Swing controls for the Java3DWorld1. 210

5 Case Study: Java 2D GUI Application with Design Patterns

5.1 Deitel Drawing application showing randomly drawn shapes (Exercise 5.8) and a ZoomDrawingView (Fig. 5.13). 223
5.2 Large-scale view of drawing from Fig. 5.1. 223
5.3 MyShape abstract base class for drawing objects. 224
5.4 MyLine subclass of class MyShape that represents a line. 230
5.5 MyRectangle subclass of class MyShape that represents a rectangle. 232
5.6 MyOval subclass of class MyShape that represents an oval. 234
5.7 MyText subclass of class MyShape that represents a string of text. 235
5.8 MyImage subclass of class MyShape that represents a JPEG image in a drawing. 240
5.9 DrawingModel Observable class that represents a drawing containing multiple MyShapes. 243
5.10 DrawingFileReaderWriter utility class for saving drawings to files and loading drawings from files. 245
5.11 Sample XML document generated by DrawingFileReaderWriter. 252
5.12 **DrawingView** class for displaying *MyShapes* in a **DrawingModel**.

5.13 **ZoomDrawingView** subclass of **DrawingView** for displaying scaled *MyShapes*.

5.14 **MyShapeController** abstract base class for controllers that handle mouse input.

5.15 **BoundedShapeController** **MyShapeController** subclass for controlling *MyLines*, *MyOvals* and *MyRectangles*.

5.16 **MyLineController** **MyShapeController** subclass for drawing *MyLines*.

5.17 **MyTextController** **MyShapeController** subclass for adding *MyText* instances to a drawing.

5.18 **MyShapeControllerFactory** class for creating appropriate **MyShapeController** for given *MyShape* type.

5.19 **DragAndDropController** for moving *MyShapes* between drawings and adding JPEG images to drawings using drag and drop.

5.20 **DragSourceListener** interface methods and their descriptions.

5.21 **DropTargetListener** interface methods and their descriptions.

5.22 **TransferableShape** enables **DragAndDropController** to transfer *MyShape* objects through drag-and-drop operations.

5.23 **DrawingInternalFrame** class that provides a user interface for creating drawings.

5.24 **DrawingFileFilter** is a **FileFilter** subclass that enables users to select Deitel Drawing files from **JFileChooser** dialogs.

5.25 **ZoomDialog** for displaying **DrawingModels** in a scalable view.

5.26 **AbstractDrawingAction** abstract base class for **Actions**.

5.27 **GradientIcon** implementation of interface **Icon** that draws a gradient.

5.28 **DeitelDrawing** application that uses a multiple-document interface for displaying and modifying **DeitelDrawing** drawings.

5.29 **SplashScreen** class for displaying a logo while the application loads.

---

6. **JavaBeans Component Model**

6.1 Forte for Java Community Edition 2.0.

6.2 **Install New JavaBean...** menu item.

6.3 **Install JavaBean** dialog.

6.4 **Select JavaBean** and **Palette Category** dialogs.

6.5 **Beans** tab in the **Component Palette** and tooltip for **LogoAnimator** JavaBean.

6.6 **Filesystems** tab in the **Explorer** window.

6.7 **Development** directory selected in **Explorer** window.

6.8 **New...** menu item.

6.9 **New...- Template Chooser** dialog.

6.10 **GUI Editing** tab of Forte.

6.11 **Component Inspector** and **Form** windows.

6.12 **Source Editor** window.

6.13 **Beans** tab of the **Component Palette**.

6.14 **LogoAnimator** icon.

6.15 **LogoAnimator** animation in the **Form** window.
6.16 Component Inspector with LogoAnimator Properties sheet.
6.17 Component Inspector drop down-menu for the background property.
6.18 Changing background color of LogoAnimator.
6.19 AnimationWindow selected in Explorer.
6.20 Selecting FlowLayout in the Explorer menu.
6.21 Swing tab of the Component Palette.
6.22 JButton icon in the Component Palette.
6.23 Adding a JButton to AnimationWindow.
6.24 Editing text property of JButton.
6.25 Component Palette Selection mode.
6.26 Component Palette Connection mode.
6.27 Select Connection mode.
6.28 Connecting JButton and LogoAnimator.
6.29 Connection Wizard dialog.
6.30 Select actionPerformed event.
6.31 Selecting method startAnimation for the target component.
6.32 Select Execute from Explorer menu.
6.33 AnimationWindow running in Forte.
6.34 Definition of class LogoAnimator.
6.35 Compile option in the Source Editor menu.
6.36 Method file manifest.tmp for the LogoAnimator bean.
6.37 Add images directory to LogoAnimator.jar.
6.38 Manifest tab of JAR Packager dialog.
6.39 LogoAnimator2 with property animationDelay.
6.40 LogoAnimator2 bean with property animationDelay exposed in Forte’s Component Inspector.
6.41 Definition for class SliderFieldPanel.
6.42 Manifest file for the SliderFieldPanel JavaBean.
6.43 Change properties currentValue and maximumValue.
6.44 Select propertyChange event.
6.45 Select animationDelay property of LogoAnimator2.
6.46 Select currentValue Bound Property.
6.47 JFrame with LogoAnimator2 and SliderFieldPanel.
6.48 ColorEvent custom-event class indicating a color change.
6.49 ColorListener interface for receiving colorChanged notifications.
6.50 Definition of class ColorSliderPanel.
6.51 Manifest file for the ColorSliderPanel JavaBean.
6.52 Selecting colorChanged method in Connection Wizard.
6.53 Selecting setBackground method for target LogoAnimator2.
6.54 Entering user code in Connection Wizard.
6.55 Using the ColorSliderPanel to change the background color of LogoAnimator2.
6.56 SliderFieldPanelBeanInfo exposes properties and events for SliderFieldPanel.
6.57 Properties and events exposed by SliderFieldPanelBeanInfo.
6.58 MaximumValueEditor is a PropertyEditor for SliderFieldPanel’s maximumValue property.
6.59 **MinimumValueEditor** is a **PropertyEditor** for *SliderFieldPanel’s minimumValue* property.

6.60 **MaximumValueEditor** and **MinimumValueEditor** pull-down menus in Forte.

6.61 **SliderFieldPanel** values constrained by **PropertyEditors**.

6.62 **SliderFieldPanelCustomizer** custom GUI for modifying **SliderFieldPanel** beans.

6.63 Select **Customize** from **Component Inspector** menu.

6.64 **SliderFieldPanel**’s **Customizer Dialog**.

7 **Security**

7.1 Encrypting and decrypting a message using a symmetric secret key.

7.2 Distributing a session key with a key distribution center.

7.3 Encrypting and decrypting a message using public-key cryptography.

7.4 Authentication with a public-key algorithm

7.5 Creating a digital envelope.

7.6 **EncipherDecipher** application for demonstrating Password-Based Encryption.

7.7 **EncipherDecipher** before and after encrypting contents.

7.8 A portion of the VeriSign digital certificate. (Courtesy of VeriSign, Inc.)

7.9 Some permissions available in the Java 2 security model.

7.10 **AuthorizedFileWriter** writes to file using a security manager.

7.11 Policy file grants permission to write to file **authorized.txt**.

7.12 Policy file grants permission to the specified codebase.

7.13 Applet that browses a user’s local filesystem.

7.14 File listing for **FileTreeApplet.jar**.

7.15 HTML file for **FileTreeApplet**.

7.16 Java Plug-in security warning when loading a signed applet.

7.17 **FileTreeApplet** browsing the *D:\jdk1.3.1\* directory.

7.18 **AuthenticateNT** uses the **NTLoginModule** to authenticate a user and invoke a **PrivilegedAction**.

7.19 **WriteFileAction** is a **PrivilegedAction** for writing a simple text file.

7.20 Configuration file for authentication using **NTLoginModule**.

7.21 JAAS policy file for granting permissions to a **Principal** and codebase.

7.22 Policy file for JAAS application.

7.23 **LoginServer** uses an **SSLServerSocket** for secure communication.

7.24 **LoginClient** communicates with **LoginServer** via SSL.

7.25 Two sample executions of class **LoginClient**.

8 **Java Database Connectivity (JDBC)**

8.1 Relational-database structure of an **Employee** table.

8.2 Result set formed by selecting **Department** and **Location** data from the **Employee** table.

8.3 **authors** table from **books**.

8.4 Data from the **authors** table of **books**.
8.5 publishers table from books.
8.6 Data from the publishers table of books.
8.7 authorISBN table from books.
8.8 Data from the authorISBN table of books.
8.9 titles table from books.
8.10 Data from the titles table of books.
8.11 Table relationships in books.
8.12 SQL query keywords.
8.13 authorID and lastName from the authors table.
8.14 Titles with copyrights after 1999 from table titles.
8.15 Authors whose last name starts with D from the authors table.
8.16 The only author from the authors table whose last name contains i as the second letter.
8.17 Authors from table authors in ascending order by lastName.
8.18 Authors from table authors in descending order by lastName.
8.19 Authors from table authors in ascending order by lastName and by firstName.
8.20 Books from table titles whose title ends with How to Program in ascending order by title.
8.21 Authors and the ISBN numbers for the books they have written in ascending order by lastName and firstName.
8.22 Table Authors after an INSERT INTO operation to add a record.
8.23 Table authors after an UPDATE operation to change a record.
8.24 Table authors after a DELETE operation to remove a record.
8.26 Displaying the authors table from the books database.
8.27 JDBC driver types.
8.28 ResultSetTableModel enables a JTable to display the contents of a ResultSet.
8.29 ResultSet constants for specifying ResultSet type.
8.30 ResultSet constants for specifying result set properties.
8.31 DisplayQueryResults for querying database books.
8.32 Table relationships in database addressbook.
8.33 AddressBookEntry bean represents an address book entry.
8.34 AddressBookDataAccess interface describes the methods for accessing the addressbook database.
8.35 DataAccessException is thrown when there is a problem accessing the data source.
8.36 CloudscapeDataAccess implements interface AddressBookDataAccess to perform the connection to the database and the database interactions.
8.37 AddressBookEntryFrame for viewing and editing an AddressBookEntry.
8.38 AddressBook application class that enables the user to interact with the addressbook database.
8.39 Screen captures of the AddressBook application.
8.40  **Statement** and **PreparedStatement** methods for batch updates. 515
8.41  Return values of method **executeBatch**. 516
8.42  **Statement** methods that enable processing of multiple results returned by method **execute**. 517

**9  Servlets**
9.1  Methods of interface **Servlet** (package **javax.servlet**). 535
9.2  Other methods of class **HttpServlet**. 537
9.3  Some methods of interface **HttpServletRequest**. 537
9.4  Some methods of interface **HttpServletResponse**. 538
9.5  **WelcomeServlet** that responds to a simple HTTP get request. 540
9.6  HTML document in which the form’s action invokes **WelcomeServlet** through the alias welcome1 specified in **web.xml**. 54
9.7  Tomcat documentation home page. (Courtesy of The Apache Software Foundation.) 544
9.8  Web application standard directories. 545
9.9  Deployment descriptor (**web.xml**) for the advjhtp1 Web application. 546
9.10  Web application directory and file structure for **WelcomeServlet**. 548
9.11  **WelcomeServlet2** responds to a get request that contains data. 549
9.12  HTML document in which the form’s action invokes **WelcomeServlet2** through the alias welcome2 specified in **web.xml**. 551
9.13  Deployment descriptor information for servlet **WelcomeServlet2**. 552
9.14  **WelcomeServlet3** responds to a post request that contains data. 553
9.15  HTML document in which the form’s action invokes **WelcomeServlet3** through the alias welcome3 specified in **web.xml**. 554
9.16  Deployment descriptor information for servlet **WelcomeServlet3**. 555
9.17  Redirecting requests to other resources. 556
9.18  **RedirectServlet.html** document to demonstrate redirecting requests to other resources. 558
9.19  Deployment descriptor information for servlet **RedirectServlet**. 559
9.20  Storing user data on the client computer with cookies. 561
9.21  **CookieSelectLanguage.html** document for selecting a programming language and posting the data to the **CookieServlet**. 565
9.22  Deployment descriptor information for servlet **CookieServlet**. 568
9.23  Important methods of class **Cookie**. 568
9.24  Maintaining state information with **HttpSession** objects. 569
9.25  **SessionSelectLanguage.html** document for selecting a programming language and posting the data to the **SessionServlet**. 571
9.26  Deployment descriptor information for servlet **WelcomeServlet2**. 577
9.27  Multi-tier Web-based survey using XHTML, servlets and JDBC. 578
9.28  **Survey.html** document that allows users to submit survey responses to **SurveyServlet**. 582
9.29  Deployment descriptor information for servlet **SurveyServlet**. 584
9.30  **HttpUtils** class methods. 584

**10  JavaServer Pages (JSP)**
10.1  Using a JSP expression to insert the date and time in a Web page. 596
10.2 JSP implicit objects.
10.3 JSP escape sequences.
10.4 Scripting a JavaServer Page—welcome.jsp.
10.5 JSP standard actions.
10.6 Action <jsp:include> attributes.
10.7 Banner (banner.html) to include across the top of the XHTML document created by Fig. 10.10.
10.8 Table of contents (toc.html) to include down the left side of the XHTML document created by Fig. 10.10.
10.9 JSP clock2.jsp to include as the main content in the XHTML document created by Fig. 10.10.
10.10 JSP include.jsp Includes resources with <jsp:include>.
10.11 JSP forward1.jsp receives a firstName parameter, adds a date to the request parameters and forwards the request to forward2.jsp for further processing.
10.12 JSP forward2.jsp receives a request (from forward1.jsp in this example) and uses the request parameters as part of the response to the client.
10.13 Attributes of the <jsp:plugin> action.
10.14 An applet to demonstrate <jsp:plugin> in Fig. 10.15.
10.15 Using <jsp:plugin> to embed a Java 2 applet in a JSP.
10.16 Attributes of the <jsp:useBean> action.
10.17 Rotator bean that maintains a set of advertisements.
10.18 JSP adrotator.jsp uses a Rotator bean to display a different advertisement on each request to the page.
10.19 Attributes of the <jsp:setProperty> action.
10.20 GuestBean stores information for one guest.
10.21 GuestDataBean performs database access on behalf of guestBookLogin.jsp.
10.22 JavaServer page guestBookLogin.jsp enables the user to submit a first name, a last name and an e-mail address to be placed in the guest book.
10.23 JavaServer page guestBookView.jsp displays the contents of the guest book.
10.24 JavaServer page guestBookErrorPage.jsp responds to exceptions in guestBookLogin.jsp and guestBookView.jsp.
10.25 JSP guest book sample output windows.
10.26 JSP directives.
10.27 Attributes of the page directive.
10.28 JSP includeDirective.jsp demonstrates including content at translation-time with directive include.
10.29 Attributes of the taglib directive.
10.30 JSP customTagWelcome.jsp uses a simple custom tag.
10.31 WelcomeTagHandler custom tag handler.
10.32 Custom tag library descriptor file advjhtpl-taglib.tld.
10.33 Specifying attributes for a custom tag.
10.34 Welcome2TagHandler custom tag handler for a tag with an attribute.
10.35 Element tag for the *welcome2* custom tag.
10.36 Using a custom tag that interacts with its body.
10.37 *GuestBookTag* custom tag handler.
10.38 *GuestBookTagExtraInfo* used by the container to define scripting variables in a JSP that uses the *guestlist* custom tag.
10.39 Element tag for the *guestlist* custom tag.

11 **Case Study: Servlet and JSP Bookstore**

11.2 Servlet and JSP components for bookstore case study.
11.3 Bookstore home page (*index.html*).
11.4 Shared cascading style sheet (*styles.css*) used to apply common formatting across XHTML documents rendered on the client.
11.5 *TitlesBean* for obtaining book information from the *books* database and creating an *ArrayList* of *BookBean* objects.
11.6 *BookBean* that represents a single book’s information and defines the XML format of that information.
11.7 JSP *books.jsp* returns to the client an XHTML document containing the book list.
11.8 *BookServlet* obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client.
11.9 XSL style sheet (*books.xsl*) that transforms a book’s XML representation into an XHTML document.
11.10 *CartItemBeans* contain a *BookBean* and the *quantity* of a book in the shopping cart.
11.11 *AddToCartServlet* places an item in the shopping cart and invokes *viewCart.jsp* to display the cart contents.
11.12 JSP *viewCart.jsp* obtains the shopping cart and outputs an XHTML document with the cart contents in tabular format.
11.13 Order form (*order.html*) in which the user inputs name, address and credit-card information to complete an order.
11.14 JSP *process.jsp* performs the final order processing.
11.15 *Application Deployment Tool* main window.
11.16 *New Application* window.
11.17 *Application Deployment Tool* main window after creating a new application.
11.18 *New Web Component Wizard - Introduction* window.
11.20 *Add Files to .WAR - Add Content Files* window.
11.21 *Add Files to .WAR - Add Class Files* window.
11.22 *Choose Root Directory* window.
11.23 *Add Files to .WAR - Add Class Files* window after selecting the root directory in which the files are located.
11.24 *New Web Component Wizard - WAR File General Properties* window after selecting the file *BookServlet.class*. 
11.25 New Web Component Wizard - Choose Component Type window. 703
11.26 New Web Component Wizard - Component General Properties window. 703
11.27 New Web Component Wizard - Component Aliases window. 703
11.28 Application Deployment Tool window after deploying BookServlet and AddToCartServlet. 704
11.29 Add Files to .WAR - Add Content Files window. 705
11.30 Add Files to .WAR - Add Class Files window. 706
11.31 Specifying the Web Context in the Application Deployment Tool. 707
11.32 Specifying the Resource Ref’s in the Application Deployment Tool. 707
11.33 Specifying the Resource Ref’s in the Application Deployment Tool. 708
11.34 Specifying the welcome file in the File Ref’s tab of the Application Deployment Tool. 708
11.35 Application Deployment Tool toolbar buttons for updating application files and deploying applications. 709
11.36 Deploy JSP and Servlet Bookstore - Introduction window. 709

12 Java-Based Wireless Applications Development and J2ME
12.1 Three-tier architecture for Tip Test. 718
12.2 Database contents of tips.sql. 718
12.3 Class WelcomeServlet sends an introductory screen that provides game directions to a client. 720
12.4 Interface ClientUserAgentHeaders contains unique User-Agent header substrings for all clients. 723
12.5 WelcomeServlet output (index.html) for XHTML client. 723
12.6 WelcomeServlet output (index.wml) for WAP client. (Image of UP.SDK courtesy Openwave Systems Inc. Openwave, the Openwave logo, and UP.SDK are trademarks of Openwave Systems Inc. All rights reserved.) 724
12.7 WelcomeServlet output (index.html) for i-mode client. (Courtesy of Pixo, Inc.) 725
12.8 WelcomeServlet output (index.txt) for J2ME client. (Courtesy of Sun Microsystems, Inc.) 725
12.9 TipTestServlet handles game logic and sends Tip Test to clients. 726
12.10 XHTMLTipQuestion.xsl transforms XML Tip-Test question to XHTML document. 741
12.11 Internet Explorer Tip-Test question output screen. 743
12.12 XHTMLTipAnswer.xsl transforms XML Tip-Test answer to XHTML document. 744
12.13 Internet Explorer Tip-Test answer output screen. 746
12.14 WAPTipQuestion.xsl transforms XML Tip-Test question to WML document. 746
12.15 Openwave UP simulator Tip-Test question screen. (Image of UP.SDK courtesy Openwave Systems Inc. Openwave, the Openwave logo, and UP.SDK are trademarks of Openwave Systems Inc. All rights reserved.) 748
12.16 WAPTipAnswer.xsl transforms answer to WML document. 749
12.17 Openwave UP simulator Tip-Test answer screen. (Image of UP.SDK courtesy Openwave Systems Inc. Openwave, the Openwave logo, and UP.SDK are trademarks of Openwave Systems Inc. All rights reserved.)
12.18 IMODETipQuestion.xsl transforms XML Tip-Test question to cHTML document.
12.19 Pixo i-mode browser Tip-Test question screen. (Courtesy of Pixo, Inc.)
12.20 IMODETipAnswer.xsl transforms XML Tip-Test answer to cHTML document.
12.21 Pixo i-mode browser Tip-Test answer screen. (Courtesy of Pixo, Inc.)
12.22 J2ME client Tip-Test question screen. (Courtesy of Sun Microsystems, Inc.)
12.23 J2ME client Tip-Test answer screen. (Courtesy of Sun Microsystems, Inc.)
12.24 J2ME java.io, java.lang and java.util packages.
12.25 MIDP javax.microedition.lcdui and javax.microedition.io packages.
12.26 MIDP javax.microedition.rms and javax.microedition.midlet packages.
12.27 TipTestMIDlet downloads Tip Test from TipTestServlet.
12.28 J2ME user-interface API class hierarchy.
12.29 TipTestMIDlet main screen. (Courtesy of Sun Microsystems, Inc.)
12.30 TipTestMIDlet welcome screen. (Courtesy of Sun Microsystems, Inc.)
12.31 TipTestMIDlet information screen. (Courtesy of Sun Microsystems, Inc.)
12.32 TipTestMIDlet Tip-Test question screen. (Courtesy of Sun Microsystems, Inc.)
12.33 TipTestMIDlet Tip-Test answer screen. (Courtesy of Sun Microsystems, Inc.)
12.34 Deployment descriptor to run WelcomeServlet and TipTestServlet.
12.35 Case-study browser URLs.

13 Remote Method Invocation
13.1 WeatherService interface.
13.2 WeatherServiceImpl class implements remote interface WeatherService.
13.3 WeatherBean stores weather forecast for one city.
13.4 WeatherServiceClient client for WeatherService remote object.
13.5 WeatherListModel is aListModel implementation for storing weather information.
13.6 WeatherCellRenderer is a custom ListCellRenderer for displaying WeatherBeans in a JList.
13.7 WeatherItem displays weather information for one city.
13.8 Running the rmiregistry.
13.9 Executing the WeatherServiceImpl remote object.
13.10 WeatherServiceClient application window.
13.11 Participants of DeitelMessenger case study.
13.12 ChatServer remote interface for Deitel Messenger chat server.
13.13 **StoppableChatServer** remote interface for stopping a **ChatServer** remote object.
13.14 **ChatServerImpl** implementation of remote interfaces **ChatServer** and **StoppableChatServer** as **Activatable** remote objects.
13.15 **ChatServerAdministrator** application for starting and stopping the **ChatServer** remote object.
13.16 Policy file for **ChatServer**'s **ActivationGroup**.
13.17 **ChatClient** remote interface to enable RMI callbacks.
13.18 **ChatMessage** is a serializable class for transmitting messages over RMI.
13.19 **MessageManager** interface for classes that implement communication logic for a **ChatClient**.
13.20 **RMIMessageManager** remote object and **MessageManager** implementation for managing **ChatClient** communication.
13.21 **MessageListener** interface for receiving new messages.
13.22 **DisconnectListener** interface for receiving server disconnect notifications.
13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client.
13.24 **DeitelMessenger** launches a chat client using classes **ClientGUI** and **RMIMessageManager**.
13.25 Policy file for the RMI activation daemon.
13.26 File listing for the HTTP server's download directory.
13.27 Policy file for **ChatServerAdministrator**.
13.28 Policy file for the **DeitelMessenger** client.
13.29 Sample conversation using Deitel Messenger.

### 14 Session EJ Bs and Distributed Transactions

14.1 Methods of interface **javax.ejb.EJBObject**.
14.2 Methods of interface **javax.ejb.EJBHome**.
14.3 **InterestCalculator** remote interface for calculating simple interest.
14.4 **InterestCalculatorHome** interface for creating **InterestCalculator** EJBs.
14.5 **InterestCalculatorEJB** implementation of **InterestCalculator** remote interface.
14.6 **InterestCalculatorClient** for interacting with **InterestCalculator** EJB.
14.7 Creating **New Application** in **Application Deployment Tool**.
14.8 Specifying EAR file for **New Application**.
14.9 Creating a **New Enterprise Bean**.
14.10 Adding **InterestCalculator** EJB classes.
14.11 Selecting **InterestCalculator** EJB classes to add.
14.12 Result of adding **InterestCalculator** EJB classes.
14.13 Specifying **Enterprise Bean Class** for **InterestCalculator** EJB.
14.14 Specifying **InterestCalculator** EJB classes and **Stateful Session Bean Type**.
14.15 Specifying **Container Managed Transactions** for **InterestCalculator** EJB.
| 14.16 | XML deployment descriptor for **InterestCalculator** EJB. | 867 |
| 14.17 | Specifying **JNDI Name** for **InterestCalculator** EJB. | 867 |
| 14.18 | Deploying enterprise application to **localhost**. | 868 |
| 14.19 | Specifying the **Application Deployment Tool** should **Return Client Jar**. | 868 |
| 14.20 | Successful completion of deployment process. | 869 |
| 14.21 | **MathTool** remote interface for calculating factorials and generating Fibonacci series. | 870 |
| 14.22 | **MathToolEJB** implementation of **MathTool** remote interface. | 870 |
| 14.23 | **MathToolHome** interface for creating **MathTool** EJBs. | 873 |
| 14.24 | **MathToolClient** for interacting with **MathTool** EJB. | 874 |
| 14.25 | **MoneyTransfer** remote interface for transferring money and getting account balances. | 880 |
| 14.26 | **MoneyTransferHome** interface for creating **MoneyTransfer** EJBs. | 880 |
| 14.27 | **MoneyTransferEJB** implementation of **MoneyTransfer** remote interface using bean-managed transaction demarcation. | 881 |
| 14.28 | **MoneyTransferEJB** implementation of **MoneyTransfer** remote interface using container-managed transaction demarcation. | 887 |
| 14.29 | Transaction types for container-managed transaction demarcation. | 891 |
| 14.30 | **MoneyTransferEJBClient** for interacting with **MoneyTransfer** EJB. | 892 |
| 14.31 | **Resource References** dialog of **New Enterprise Bean Wizard**. | 898 |
| 14.32 | Add **Resource Reference** for BankABC. | 899 |
| 14.33 | Add **Resource Reference** for BankXYZ. | 899 |
| 14.34 | Selecting **Bean-Managed Transactions**. | 900 |
| 14.35 | Selecting **Container-Managed Transactions**. | 900 |

### 15 Entity EJBs

| 15.1 | **Employee** remote interface for setting and getting **Employee** information. | 906 |
| 15.2 | **EmployeeHome** interface for finding and creating **Employee** EJBs. | 908 |
| 15.3 | **EmployeeEJB** implementation of **Employee** remote interface using bean-managed persistence. | 909 |
| 15.4 | **General** dialog of **New Enterprise Bean Wizard**. | 919 |
| 15.5 | **Bean-Managed Persistence** selected in **Entity Settings** dialog. | 919 |
| 15.6 | **Resource References** dialog in **New Enterprise Bean Wizard**. | 920 |
| 15.7 | **EmployeeEJB** implementation of **Employee** remote interface using container-managed persistence. | 921 |
| 15.8 | **Container-Managed Persistence** selected in **Entity Settings** dialog. | 924 |
| 15.9 | **EmployeeEJBClient** for interacting with **Employee** EJB. | 926 |

### 16 Messaging with JMS

| 16.1 | Setting environment variables for J2EE 1.3 installation. | 939 |
| 16.2 | Point-to-point messaging model. | 940 |
| 16.3 | **Voter** application overview. | 940 |
| 16.4 | **Voter** class submits votes as messages to queue. | 941 |
| 16.5 | **Voter** application votes for favorite programming language | 945 |
| 16.6 | **VoteCollector** class retrieves and tallies votes. | 945 |
| 16.7 | **VoteCollector** tallies and displays votes. | 948 |
| 16.8 | **VoteListener** class receives messages from the queue. | 949 |
16.9 **TallyPanel** class displays candidate name and tally. 950
16.10 Publish/subscribe messaging model. 952
16.11 Weather application overview. 952
16.12 **WeatherPublisher** class publishes messages to **Weather** topic. 953
16.13 **WeatherPublisher** publishing weather update messages. 958
16.14 **WeatherSubscriber** class allows user to receive weather updates. 958
16.15 **WeatherSubscriber** selecting cities for weather updates. 963
16.16 **WeatherSubscriber** having received updated weather conditions. 964
16.17 **WeatherListener** class subscribes to **Weather** topic to receive weather forecasts. 964
16.18 **WeatherDisplay** displays **WeatherBeans** in a **JList** using a | 965
   **WeatherCellRenderer**.
16.19 **Voter** application overview. 968
16.20 **CandidateHome** interface for **Candidate** EJB. 969
16.21 **Candidate** remote interface for **Candidate** EJB. 969
16.22 **CandidateEJB** class to maintain candidate tallies. 970
16.23 **VoteCollectorEJB** class tallies votes from **Votes** queue. 972
16.24 **TallyDisplay** displays candidate tallies from database. 975
16.25 **TallyDisplay** displays candidate tallies from database. 977
16.26 **TallyPanel** class displays the name and tally for a candidate. 977
16.27 EJB JAR settings for **VoteCollectorApp** application. 979
16.28 Add class files for **Candidate** EJB. 979
16.29 General settings for **Candidate** EJB. 980
16.30 Entity settings for **Candidate** EJB. 980
16.31 Entity tab for **Candidate** EJB. 981
16.32 Database settings for **Candidate** EJB. 981
16.33 SQL generation for **Candidate** EJB. 982
16.34 SQL warning for **Candidate** EJB. 982
16.35 EJB JAR settings for **VoteCollector** EJB. 983
16.36 Add class file for **VoteCollector** EJB. 983
16.37 General settings for **VoteCollector** EJB. 984
16.38 Transaction management settings for the **VoteCollector** EJB. 984
16.39 Message-Driven Bean settings for **VoteCollector** EJB. 985
16.40 Enterprise Bean References for **VoteCollector** EJB. 985
16.41 Setting JNDI names for **VoteCollectorApp**. 986
16.42 Deploying the **VoteCollector** application. 986

**17**  **Enterprise Java Case Study: Architectural Overview**
17.1 Three-tier application model in Deitel Bookstore. 992
17.2 Detailed architecture of Deitel Bookstore Enterprise Java case study. 994
17.3 XML file generated by **GetProductServlet**. 996
17.4 XSL transformation for generating XHTML from **GetProductServlet**. 996
17.5 XHTML document generated by XSLT in **GetProductServlet**. 998
17.6 XSL transformation for generating WML from **GetProductServlet**. 1000
17.7 WML document generated by XSLT in **GetProductServlet**. 1000
   (Image © 2001 Nokia Mobile Phones.) 1001
17.8 XSL transformation for generating cHTML from **GetProductServlet**. 1003
17.9 cHTML document generated by XSLT in `GetProductServlet`.  
(Image courtesy of Pixo, Inc.)  

18  **Enterprise Java Case Study: Presentation and Controller Logic**  
18.1 `XMLServlet` base class for servlets in the Deitel Bookstore.  
18.2 Configuration file for enabling support for various client types  
(`clients.xml`).  
18.3 DTD for `clients.xml`.  
18.4 `ClientModel` for representing supported clients.  
18.5 Flow of client requests and data returned in the Deitel Bookstore  
for XHTML clients.  
18.6 `AddToCartServlet` for adding products to a shopping cart.  
18.7 `ViewCartServlet` for viewing contents of shopping cart.  
18.8 `ViewCartServlet` XSL transformation for XHTML browsers  
(`XHTML/viewCart.xsl`).  
18.9 `ViewCartServlet` XSL transformation for i-mode browsers  
(`cHTML/viewCart.xsl`). (Image courtesy of Pixo, Inc.)  
18.10 `ViewCartServlet` XSL transformation for WML browsers  
(`WML/viewCart.xsl`). (Image © 2001 Nokia Mobile Phones.)  
18.11 `RemoveFromCartServlet` for removing products from shopping cart.  
18.12 `UpdateCartServlet` for updating quantities of products in  
shopping cart.  
18.13 `CheckoutServlet` for placing `Orders`. (Images courtesy Pixo,  
Inc. or © 2001 Nokia Mobile Phones.)  
18.14 `GetAllProductsServlet` for viewing the product catalog.  
(Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)  
18.15 `GetProductServlet` for viewing product details. (Images courtesy  
Pixo, Inc. or © 2001 Nokia Mobile Phones.)  
18.16 `ProductSearchServlet` for searching product catalog. (Images  
courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)  
18.17 `RegisterServlet` for registering new `Customers`.  
18.18 `LoginServlet` for authenticating registered `Customers`.  
(Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)  
18.19 `ViewOrderHistoryServlet` for viewing customer’s previously placed  
`Orders`. (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)  
18.20 `ViewOrderServlet` for viewing details of an order. (Images courtesy  
Pixo, Inc. or © 2001 Nokia Mobile Phones.)  
18.21 `GetPasswordHintServlet` for viewing a `Customer’s` password  
hint. (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)  

19  **Enterprise Java Case Study: Business Logic Part 1**  
19.1 Communication between `GetProductServlet` and `Product EJB`.  
19.2 `ShoppingCart` remote interface for adding, removing and updating  
`Products`, checking out and calculating the `Order’s` total cost.  
19.3 `ShoppingCartEJB` implementation of `ShoppingCart` remote  
interface.
19.4 **ShoppingCartHome** interface for creating **ShoppingCart** EJB instances.
19.5 **ShoppingCart** general deployment settings.
19.6 **ShoppingCart** EJB references.
19.7 **Product** remote interface for modifying details of **Product** EJB instances.
19.8 **ProductEJB** implementation of **Product** remote interface.
19.9 **ProductHome** interface for finding and creating **Product** EJB instances.
19.10 **ProductModel** class for serializing **Product** data.
19.11 **XMLGenerator** interface for generating XML Elements for public properties.
19.12 **Product** general deployment settings.
19.13 **Product Entity** and deployment settings.
19.14 **Order** remote interface for modifying details of **Order** EJB instances.
19.15 **OrderEJB** implementation of **Order** remote interface.
19.16 **OrderHome** interface for finding and creating **Order** EJB instances.
19.17 **OrderModel** class for serializing **Order** data.
19.18 **Order** general deployment settings.
19.19 **Order** entity and deployment settings.
19.20 **Order** EJB references.
19.21 **OrderProduct** remote interface for modifying details of **OrderProduct** EJB instances.
19.22 **OrderProductEJB** implementation of **OrderProduct** remote interface.
19.23 **OrderProductHome** interface for finding and creating **OrderProduct** EJB instances.
19.24 **OrderProductPK** primary-key class for **OrderProduct** EJB.
19.25 **OrderProductModel** class for serializing **OrderProduct** data.
19.26 **OrderProduct** general deployment settings.
19.27 **OrderProduct** entity and deployment settings.
19.28 **OrderProduct** EJB references.

20 **Enterprise Java Case Study: Business Logic Part 2**
20.1 **Customer** remote interface for modifying **Customer** details, getting an **Order** history and password hint.
20.2 **CustomerEJB** implementation of **Customer** remote interface.
20.3 **CustomerHome** interface for creating and finding **Customer** EJB instances.
20.4 **CustomerModel** for serializing **Customer** data.
20.5 **Customer** general deployment settings.
20.6 **Customer** entity and deployment settings.
20.7 **Customer** EJB References.
20.8 **Address** remote interface for modifying **Address** details.
20.9 **AddressEJB** implementation of **Address** remote interface.
20.10 **AddressHome** interface for creating and finding **Address** EJB instances.
20.11 **AddressModel** for serializing **Address** EJB data.
20.12 **Address General** deployment settings.
20.13 **Address** entity and deployment settings.  
20.14 **Address** EJB references.  
20.15 **SequenceFactory** remote interface for generating primary keys.  
20.16 **SequenceFactoryEJB** implementation of **SequenceFactory** remote interface.  
20.17 **SequenceFactoryHome** interface for finding **SequenceFactory** EJB instances.  
20.18 **SequenceFactory** general deployment settings.  
20.19 **SequenceFactory** entity and deployment settings.  
20.20 Adding an EJB to an enterprise application.  
20.21 Creating an EJB JAR file.  
20.22 Specifying the **Root Directory** for EJB classes.  
20.23 Adding EJB classes to an EJB JAR file.  
20.24 Results of adding EJB classes to an EJB JAR file.  
20.25 Specifying classes for EJB, home interface and remote interface.  
20.26 Setting **Bean Type** to **Entity**.  
20.27 Configuring container-managed fields and primary-key class.  
20.28 Specifying other EJBs referenced by this EJB.  
20.29 Specifying **Container-Managed Transactions** for EJB business methods.  
20.30 XML descriptor generated by **Application Deployment Tool**.  
20.31 Specifying EJB **Deployment Settings**.  
20.32 Configuring EJB **Database Settings**.  
20.33 Dialog indicating methods that require **WHERE** clauses for SQL queries.  
20.34 Specifying SQL query for method **findByCustomerID**.  
20.35 Deployment settings for Deitel Bookstore servlets.  
20.36 Setting the **Context Root** for the Deitel Bookstore servlets.  
20.37 Setting the **CLIENT_LIST Context Parameter** for the Deitel Bookstore servlets.  
20.38 Servlet **EJB References**.  

### 21 Application Servers

21.1 Application server required APIs.  
21.2 WebLogic administration console. (Courtesy BEA Systems.)  
21.3 JDBC Connection pool properties. (Courtesy of BEA Systems, Inc.)  
21.4 **Weblogic-ejb-jar.xml** defines WebLogic deployment properties for Bookstore case study.  
21.5 Optional tags for **weblogic-ejb-jar.xml** not used in text.  
21.6 **Weblogic-cmp-rdbms-jar-address.xml** defines WebLogic CMP database properties for EJB **Address**.  
21.7 **WebLogic-cmp-rdbms-jar-Customer.xml** defines WebLogic CMP database properties for EJB **CustomerEJB**.  
21.8 Some WebLogic Query Language operations and examples.  
21.9 **Weblogic-cmp-rdbms-jar-order.xml** defines WebLogic CMP database properties for EJB **OrderEJB**.
21.10 Weblogic-cmp-rdbms-jar-orderProduct.xml defines WebLogic CMP database properties for the OrderProduct EJB. 1185
21.11 weblogic-cmp-rdbms-jar-product.xml defines WebLogic CMP database properties for the Product EJB. 1187
21.12 Weblogic-cmp-rdbms-jar-sequence.xml defines WebLogic CMP database properties for the SequenceFactory EJB. 1189
21.13 Weblogic.xml Web application deployment descriptor. 1190
21.14 WHERE clauses for bookstore finder methods. 1193

22 Jini
22.1 StartService window. 1200
22.2 WebServer configuration tab. 1200
22.3 RMID configuration tab. 1201
22.4 Specifying the RMID log directory. 1201
22.5 Reggie lookup service configuration tab. 1202
22.6 Run panel for starting and stopping Jini basic services. 1202
22.7 LookupBrowser configuration tab. 1203
22.8 LookupBrowser application window. 1204
22.9 Registrar menu for viewing computers that provide lookup services. 1204
22.10 UnicastDiscovery performs unicast discovery to locate Jini lookup services. 1205
22.11 Policy file that grants AllPermission to all code. 1208
22.12 UnicastDiscovery application output. 1209
22.13 MulticastDiscovery performs multicast discovery to locate Jini lookup services. 1210
22.14 MulticastDiscovery application output. 1213
22.15 Seminar maintains the location and title of a seminar. 1215
22.16 SeminarInterface defines the methods available from the SeminarInfo Jini service. 1216
22.17 BackEndInterface defines methods available to the SeminarInfo service proxy. 1216
22.18 SeminarProxy is a service proxy that clients use to communicate with the SeminarInfo service. 1217
22.19 SeminarInfo implements the SeminarInfo Jini service. 1218
22.20 Content of SeminarInfo.txt. 1220
22.21 SeminarInfoService registers the SeminarInfo service with lookup services. 1220
22.22 UnicastSeminarInfoClient is a client for the SeminarInfo service. 1224
22.23 SeminarService.jar contents. 1228
22.24 SeminarClient.jar contents. 1230
22.25 SeminarServiceDownload.jar contents. 1230
22.26 Web server configuration for SeminarInfo service. 1230
22.27 UnicastSeminarInfoClient application output. 1231
22.28 UnicastDiscoveryUtility uses class LookupLocatorDiscovery to facilitate lookup service discovery. 1232
22.29 Using MulticastDiscovery to obtain sample data for testing UnicastDiscoveryUtility.
UnicastDiscoveryUtility application output. 1237
GeneralDiscoveryUtility uses class LookupDiscoveryManager to perform both unicast and multicast lookup service discovery. 1238
GeneralDiscoveryUtility application output. 1242
Standard Jini Entry attributes. 1242
SeminarProvider subclass of Entry for describing the Seminar provider as a Jini attribute. 1243
SeminarInfoLeaseService uses class LeaseRenewalManager to manage SeminarInfo service leasing. 1244
SeminarServiceWithLeasing.jar contents. 1248
SeminarInfoJoinService uses class JoinManager to facilitate registering the SeminarInfo service and manage its leasing. 1249
SeminarServiceJoinManager.jar contents. 1251

23 JavaSpaces
Discovering a JavaSpaces service. 1262
AttendeeCounter is an Entry for keeping track of registrations for a seminar on a particular day. 1265
Writing an Entry into a JavaSpaces service. 1267
Results of running the WriteOperation application. 1269
Reading an Entry from JavaSpaces service. 1270
Results of running the ReadOperation application. 1273
Taking an Entry from a JavaSpaces service. 1273
Results of running the TakeOperation application. 1276
EntryListener for NotifyOperation application. 1277
Receiving notifications when matching Entries are written into JavaSpace. 1278
NotifyOperation Output samples. 1280
Removing entries from JavaSpaces service using method snapshot. 1281
SnapshotUsage Output window. 1284
UpdateInputWindow user interface. 1285
Finding Jini TransactionManager. 1288
Updating an entry using Jini TransactionManager. 1290
WriteOperation Output and UpdateInputWindow user interface. 1293
UpdateOperation Output and ReadOperation Output. 1294
Structure of the ImageProcessor distributed application. 1294
ImageEntry defines the Entries to store in the JavaSpaces service. 1295
Image processing node that uses the JavaSpaces service. 1296
Class Filters applies a Java 2D filter to an image. 1299
Image-processing distributed system client. 1302
Partitioning an image into smaller pieces and storing subimages in a JavaSpaces service. 1307
Partitioning and reforming an image. 1310
Displaying an image. 1312
GUI from ImageProcessorMain and ImageCollector applications. 1313
Images before and after blurring. 1313
## 24 Java Management Extensions (JMX)

24.1 JMX’s three-level management architecture. 1321
24.2 Architecture of case study management application. 1323
24.3 Defining the `PrinterMBean` interface that exposes the printer’s management capabilities. 1324
24.4 Defining an event listener for the printer to handle out-of-paper, low-toner, and paper-jam events. 1325
24.5 `Printer` MBean implementation class that represents the management contact point for all applications wishing to manage the printer. 1325
24.6 Printer simulation class capable of triggering three events. 1332
24.7 JMX Agent Architecture. 1339
24.8 Creating and starting a management agent. 1339
24.9 Notification broadcaster MBean interface. 1342
24.10 Notification broadcaster MBean implementation that broadcasts events generated by the printer. 1343
24.11 Receiving event notifications from the MBean server and handling the printer-specific events. 1344
24.12 Connecting to the `MBeanServer` remotely and creating a `PrinterSimulator` MBean. 1347
24.13 GUI for the management application. 1348
24.14 Initial output window. 1358
24.15 Printer status after an out-of-paper event occurred. 1358
24.16 Printer status after an add-paper action is taken. 1359
24.17 Printer status after a paper-jam event occurred. 1359
24.18 Printer status after a cancel pending print jobs action is taken. 1360

## 25 Jiro

25.1 Jiro technology three-tier management architecture. 1366
25.2 Jiro GUI: `Igniter` initial screen. 1367
25.3 GUI with `Display Console` checked after the start process is completed. 1368
25.4 `PrinterManagement` interface definition. 1370
25.5 `PrinterManagementImpl` implementation of interface `PrinterManagement`. 1371
25.6 `PrinterEventListener` used by all classes subscribed for events from `Printer`. 1379
25.7 Custom error class thrown by `Printer`. 1382
25.8 Printer simulator implementation. 1383
25.9 `PrinterManagementImpl.properties` file. 1391
25.10 Deployment results. 1395
25.11 `PrinterManagementStarter` dynamic service instantiator program. 1396
25.12 Finds dynamic service proxies within a lookup service. 1397
25.13 Management console user interface. 1400
25.14 Checking printer status. 1407
25.15 `Igniter` showing printer out-of-paper event. 1408
25.16 `PrinterClientGUI` showing printer out-of-paper event. 1408
25.17 `OutofPaperPolicy` interface. 1410
25.18 OutofPaperPolicy implementation. 1411
25.19 OutofPaperPolicyImpl.properties property file for OutofPaperPolicyImpl. 1415
25.20 Low toner policy interface. 1415
25.21 Low toner policy implementation. 1416
25.22 Property file for LowTonerPolicyImpl. 1420
25.23 Contents of PrinterManagementService.jar. 1421
25.24 Contents of PrinterManagementService-ifc.jar 1422
25.25 Contents of PrinterManagementService-ifc.jar. 1422
25.26 Contents of PrinterManagementService-impl.jar. 1423
25.27 Command line arguments for jarpackw. 1423
25.28 Command line arguments for jardeploy. 1424
25.29 Management policies instantiating utility. 1424
25.30 Igniter displaying out-of-paper event. 1426
25.31 OutofPaperPolicy handling out-of-paper event. 1427
25.32 Log contents after events handled by management policies. 1427
25.33 Detailed log information for a specified entry. 1428
25.34 Printer management solution work flow diagram. 1430

26 Common Object Request Broker Architecture (CORBA): Part 1
26.1 IDL definition for server SystemClock. 1443
26.2 A Java interface generated by idlj. 1444
26.3 SystemClockOperations interface generated by idlj. 1445
26.4 Implementation of the SystemClock server. 1445
26.5 Client that connects to SystemClock. 1449
26.6 Call path from a client to a distributed object. 1453
26.7 Object Management Architecture reference model. Courtesy of Object Management Group, Inc. 1455
26.8 ORB request-interface structure. Courtesy of Object Management Group, Inc. 1455
26.9 IDL keywords, types and their mappings to Java keywords. 1458
26.10 IDL file testing many of the IDL keywords and types. 1459
26.11 IDL-generated file StructMap.java (re-formatted for clarity). 1464
26.12 IDL-generated file InterfaceNameOperations.java (re-formatted for clarity). 1465
26.13 IDL-generated file InterfaceName.java (re-formatted for clarity). 1466
26.14 Deadlock caused by client calling a server that calls the client. 1468
26.15 alarmclock1.idl. 1468
26.16 AlarmClockImpl is the AlarmClock server implementation. 1469
26.17 ClockClientGUI informs the user when the alarm has sounded. 1472
26.18 AlarmClockClient is the AlarmClock client. 1474
26.19 A user-defined CORBA exception (DatabaseException) and an operation capable of throwing the exception. 1478
26.20 The generated DatabaseException.java file (reformatted for clarity). 1478
26.21 alarmclock2.idl is the IDL for the AlarmClock example. 1478
26.22 Excerpt from AlarmClockImpl.java. 1479
26.23 ChatServer, ChatClient and ChatMessage interface definitions. 1482
26.24 ChatServerImpl implementation of the CORBA ChatServer. 1484
26.25 CORBAMessageManager implementation of interface MessageManager using CORBA. 1489
26.26 DeitelMessenger application for launching the CORBA chat client. 1493
26.27 Chat.idl with ChatMessage changed to be a valuetype. 1496
26.28 Keywords specific to valuetypes. 1497
26.29 ChatMessageImpl is the ChatMessage implementation. 1497

27 Common Object Request Broker Architecture (CORBA): Part 2
27.1 SystemClockClient modified to support DII. 1511
27.2 Persistent State Definition Language example. 1520
27.3 Supplier-to-consumer flow using the Event/Notification Service. 1522
27.4 IDL keywords to support the CORBA Component Model. 1523
27.5 CORBA component types and descriptions. 1526
27.6 Customer component IDL definition demonstrating keywords publishes and emits for issuing events. 1527
27.7 ChatServerImpl implements the Deitel messenger ChatServer using RMI-IIOP. 1532
27.8 ChatServerAdministrator application for starting and stopping RMI-IIOP ChatServer. 1535
27.9 RMIIOPEndPointManager implements the ChatClient and MessageManager interfaces using RMI-IIOP. 1539
27.10 DeitelMessenger creates a ClientGUI and RMIIOPEndPointManager to launch the RMI-IIOP messenger client. 1541

28 Peer-to-Peer Applications and JXTA
28.1 Common P2P applications. 1550
28.2 Sample windows of Deitel Instant Messenger. 1552
28.3 Interface IMService specifies how service proxy interacts with the service. 1553
28.4 Interface IMPeer specifies interaction between peers. 1553
28.5 Class Message defines an object for sending and receiving messages between peers. 1554
28.6 IMServiceImpl service implementation for our case study. 1555
28.7 Class IMPeerListener is the GUI that starts peer communication. 1557
28.8 Class IMPeerImpl is the IMPeer implementation. 1560
28.9 Class IMServiceManager registers IMServiceImpl with lookup services. 1562
28.10 Class PeerList is the GUI for finding peers. 1564
28.11 MulticastSendingThread broadcasts DatagramPackets. 1572
28.12 Interface IMConstants defines Deitel-Instant-Messenger constants. 1575
28.13 Class MulticastReceivingThread uses threads to add and remove peers. 1577
28.14 Interface **PeerDiscoveryListener** listens for when peers are added and removed from peer groups.

28.15 Modified **PeerList** enables the use of classes **MulticastReceivingThread** and **PeerDiscoveryListener** in the Deitel Instant Messenger.

28.16 JXTA low-level protocols.

### 29 Introduction to Web Services and SOAP

29.1 Class SimpleService.

29.2 SOAP package administration tool.

29.3 Description of deployed service.

29.4 Client making a SOAP request.

29.5 SOAP implementation of class **WeatherService**.

29.6 SOAP implementation of class **WeatherServiceClient**.

29.7 **Apache SOAP Admin** page.

29.8 Apache SOAP **Service Deployment Descriptor Template**.

29.9 SOAP **WeatherService Client**.

### A Creating Markup with XML

A.1 Simple XML document containing a message.

A.2 XML document missing an end tag.


A.4 Using a **CDATA** section.

A.5 Demonstrating XML namespaces.

A.6 Using default namespaces.

### B Document Type Definition (DTD)

B.1 XML document declaring its associated DTD.

B.2 Validation by using an external DTD.

B.3 Invalid XML document.

B.4 Occurrence indicators.

B.5 Example of a mixed-content element.

B.6 Changing a pipe character to a comma in a DTD.

B.7 Declaring attributes.

B.8 XML document with **ID** and **IDREF** attribute types.

B.9 Error displayed when an invalid **ID** is referenced.

B.10 XML document that contains an **ENTITY** attribute type.

B.11 Error generated when a DTD contains a reference to an undefined entity.

B.12 Conditional sections in a DTD.

B.13 XML document that conforms to **conditional.dtd**.

B.14 Processing whitespace in an XML document.

### C Document Object Model (DOM™)

C.1 Article marked up with XML tags.

C.2 **XMLInfo** displays information about XML input.

C.3 DOM classes and interfaces.

C.4 Some **Document** methods.
### Illustrations

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.5</td>
<td>Node methods.</td>
<td>1659</td>
</tr>
<tr>
<td>C.6</td>
<td>Some node types.</td>
<td>1659</td>
</tr>
<tr>
<td>C.7</td>
<td>Element methods.</td>
<td>1659</td>
</tr>
<tr>
<td>C.8</td>
<td>Simple example that replaces an existing text node.</td>
<td>1660</td>
</tr>
<tr>
<td>C.9</td>
<td>Class definition for <strong>MyErrorHandler</strong>.</td>
<td>1663</td>
</tr>
<tr>
<td>C.10</td>
<td>Input document <em>(intro.xml)</em> and output from <strong>ReplaceText.java</strong>.</td>
<td>1664</td>
</tr>
<tr>
<td>C.11</td>
<td>Building an XML document with the DOM.</td>
<td>1665</td>
</tr>
<tr>
<td>C.12</td>
<td>Output for <strong>buildXml.java</strong>.</td>
<td>1668</td>
</tr>
<tr>
<td>C.13</td>
<td>Traversing the DOM.</td>
<td>1668</td>
</tr>
<tr>
<td>C.14</td>
<td>Sample execution of <strong>TraverseDOM.java</strong>.</td>
<td>1671</td>
</tr>
</tbody>
</table>

### D XSL: Extensible Stylesheet Language Transformations (XSLT)

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Java application that performs XSL transformations.</td>
<td>1677</td>
</tr>
<tr>
<td>D.2</td>
<td>Simple template.</td>
<td>1679</td>
</tr>
<tr>
<td>D.3</td>
<td>Sample input XML document <em>intro.xml</em>.</td>
<td>1680</td>
</tr>
<tr>
<td>D.4</td>
<td>Results of XSL transformation.</td>
<td>1680</td>
</tr>
<tr>
<td>D.5</td>
<td>XML document containing a list of sports.</td>
<td>1681</td>
</tr>
<tr>
<td>D.6</td>
<td>Using XSLT to create elements and attributes.</td>
<td>1681</td>
</tr>
<tr>
<td>D.7</td>
<td>Default XSLT templates.</td>
<td>1682</td>
</tr>
<tr>
<td>D.8</td>
<td>Output of transformation.</td>
<td>1683</td>
</tr>
<tr>
<td>D.9</td>
<td>Book table of contents as XML.</td>
<td>1684</td>
</tr>
<tr>
<td>D.10</td>
<td>Transforming XML data into XHTML.</td>
<td>1684</td>
</tr>
<tr>
<td>D.11</td>
<td>Output of the transformation.</td>
<td>1686</td>
</tr>
<tr>
<td>D.12</td>
<td>Day planner XML document.</td>
<td>1687</td>
</tr>
<tr>
<td>D.13</td>
<td>Using conditional elements.</td>
<td>1688</td>
</tr>
<tr>
<td>D.14</td>
<td>XSLT document being imported.</td>
<td>1690</td>
</tr>
<tr>
<td>D.15</td>
<td>Importing another XSLT document.</td>
<td>1691</td>
</tr>
<tr>
<td>D.16</td>
<td>Transformation results.</td>
<td>1692</td>
</tr>
<tr>
<td>D.17</td>
<td>Combining style sheets using <strong>xsl:include</strong>.</td>
<td>1692</td>
</tr>
<tr>
<td>D.18</td>
<td>XSLT document for rendering the author’s name.</td>
<td>1693</td>
</tr>
<tr>
<td>D.19</td>
<td>XSLT document for rendering chapter names.</td>
<td>1693</td>
</tr>
<tr>
<td>D.20</td>
<td>Output of an XSLT document using element <strong>include</strong>.</td>
<td>1694</td>
</tr>
<tr>
<td>D.21</td>
<td>Demonstrating <strong>xsl:variable</strong>.</td>
<td>1695</td>
</tr>
<tr>
<td>D.22</td>
<td>Result of <strong>variables.xsl</strong> transformation.</td>
<td>1695</td>
</tr>
</tbody>
</table>

### G Java Native Interface (JNI)

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.1</td>
<td><strong>JNIPrintWrapper</strong> loads a library and declares a <strong>native</strong> method.</td>
<td>1707</td>
</tr>
<tr>
<td>G.2</td>
<td><strong>JNIPrintWrapper.h</strong> header file generated by <strong>javah</strong>.</td>
<td>1708</td>
</tr>
<tr>
<td>G.3</td>
<td><strong>JNIPrintWrapperImpl.cpp</strong> implements the <strong>javah</strong> header to print a message.</td>
<td>1709</td>
</tr>
<tr>
<td>G.4</td>
<td><strong>JNIPrintMain</strong> calls the <strong>native</strong> method via the wrapper class.</td>
<td>1710</td>
</tr>
<tr>
<td>G.5</td>
<td><strong>JNIPIWrapper</strong> encapsulates the <strong>native</strong> methods and loads the library.</td>
<td>1710</td>
</tr>
<tr>
<td>G.6</td>
<td><strong>PIContainer</strong> returns the <strong>PI</strong> member of <strong>java.lang.Math</strong>.</td>
<td>1711</td>
</tr>
<tr>
<td>G.7</td>
<td><strong>JNIPIWrapper.h</strong> is the <strong>javah</strong> generated header file for the <strong>native</strong> functions.</td>
<td>1711</td>
</tr>
<tr>
<td>G.8</td>
<td><strong>JNIPIWrapperImpl.cpp</strong> demonstrates method calls and object construction.</td>
<td>1712</td>
</tr>
</tbody>
</table>
G.9 Signature type mappings.
G.10 JNIPIMain calls each native method via the wrapper class.
G.11 JNIStaticWrapper loads JNIMathLibrary and declares native method printStaticMembers.
G.12 MathConstants contains common math constants from Math.
G.13 JNIStaticWrapper.h javah generated header file.
G.14 JNIStaticWrapperImpl accesses and prints static members of the given MathConstants class.
G.15 JNIStaticMain prints static math constants via the wrapper class.
G.16 JNIArrayWrapper loads JNIArrayLibrary and displays the numbers in the returned array.
G.17 JNIArrayWrapper.h javah generated header file.
G.18 JNIArrayWrapperImpl.cpp demonstrates primitive and Object array creation and control.
G.19 JNIArray loads library and calls JNIArrayWrapper to print 10 numbers.
G.20 ImageSizeException used when image is too large.
G.21 PixelTintException is used for invalid pixel tint values.
G.22 JNITintWrapper loads the native library and wraps the native function.
G.23 JNITintWrapper.h javah generated JNI header file.
G.24 JNITintImages.cpp tints an array of sRGB color values to demonstrate exception handling.
G.25 JNIPanel creates the application GUI and calls the native method.
G.26JNIImageFrame serves as an entry point for the application.

H Career Opportunities
H.1 The Monster.com home page. (Courtesy of Monster.com.)
H.2 FlipDog.com job search. (Courtesy of Flipdog.com)
H.3 List of a job seeker’s criteria.
H.4 Advantage Hiring, Inc.’s Net-Interview™ service. (Courtesy of Advantage Hiring, Inc.)
H.5 Cruel World online career services. (Courtesy of Cruel World.)
H.6 eLance.com request for proposal (RFP) example. (Courtesy of eLance, Inc.)

I Unicode®
I.1 Correlation between the three encoding forms.
I.2 Various glyphs of the character A.
I.3 Java program that uses Unicode encoding.
I.4 Some character ranges.
Preface

Live in fragments no longer. Only connect.
Edward Morgan Forster

Welcome to Advanced Java 2 Platform How to Program and the exciting world of advanced-programming concepts with the three major Java platforms—Java™ 2 Enterprise Edition (J2EE), Java 2 Standard Edition (J2SE) and Java 2 Micro Edition (J2ME). Little did we know when we attended the November 1995 Internet/World Wide Web conference in Boston what that session would yield—four editions of Java How To Program (the world’s best-selling Java textbook), and now this book about Java software-development technologies for upper-level college courses and professional developers.

Before Java appeared, we were convinced that C++ would replace C as the dominant application-development language and systems-programming language for the next decade. However, the combination of the World Wide Web and Java now increases the prominence of the Internet in information-systems planning and implementation. Organizations want to integrate the Internet “seamlessly” into their information systems. Java is more appropriate than C++ for this purpose—as evidenced by Sun Microsystems’ announcement in 2001 that over 96% of enterprise application servers support J2EE.

Advanced Java 2 Platform How to Program is the first book in our Advanced How to Program series. We discuss Java technologies that may be unfamiliar and challenging to the average Java programmer. We structured each chapter discussion to provide the reader with an introduction to leading-edge and complex Java technologies, rather than provide a detailed analysis of every nuance of each topic. In fact, each topic we present could be a 600–800 page book in itself.

We use a different approach with the examples in this book than that of programming examples in our previous books. We provide fewer programs, but these programs are more substantial and illustrate sophisticated coding practices. We integrate many technologies to create a book for developers that enables you to “go beyond” and experiment with the most
up-to-date technologies and most widely employed design concepts. What better way to
learn than to work with actual technologies and code?

When determining the appropriate topics for this book, we read dozens of journals, reviewed the Sun Microsystems Web site and participated in numerous trade shows. We audited our material against the latest technologies presented at the JavaOne conference—the leading Java-developer conference sponsored by Sun Microsystems—and at other popular Java conferences. We also reviewed books on specialized Java topics. After this extensive research, we created an outline for this book and sent it for professional review by Java experts. We found so many topics we wanted to include that we wound up with over 1800 pages of material (several hundred of those pages appear as PDF documents on the CD that accompanies this book). We apologize if this is inconvenient, but the material and the number of topics are voluminous. We will most likely split the next edition into two volumes.

This book benefitted from an unusually large pool of excellent reviewers and the detailed documentation that Sun makes available on their Web site (www.sun.com). We were excited to have a number of reviewers from Sun and many other distinguished industry reviewers. We wanted experienced developers to review our code and discussions, so we could offer “expert advice” from people who actually work with the technologies in industry.

We are pleased to include a discussion of application servers in Chapter 21. The three most popular application server software products are BEA’s WebLogic, IBM’s WebSphere and Sun/Netscape’s iPlanet. Originally, we had planned to include all three on the book’s accompanying CD, but we have included only WebLogic and WebSphere. iPlanet was about to publish a new version as this book went to publication. By mutual agreement between iPlanet and Deitel & Associates, Inc., we decided not to include this software, but iPlanet provides a link to a site specific to this book—www.iplanet.com/ias_deitel—where readers can download the latest iPlanet software. We also include a discussion of how to deploy our case study on the iPlanet server. You can find this discussion on our Web site—www.deitel.com.

We moved four chapters from Java How to Program, Third Edition—RMI, Servlets, JavaBeans and JDBC—to Advanced Java 2 Platform How to Program. Prentice Hall has published a paperback supplement (ISBN: 0-13-074367-4) containing these four chapters for readers who have purchased Java How to Program, Fourth Edition.

The world of Java is growing so rapidly that Advanced Java 2 Platform How to Program and its companion text, Java How to Program, Fourth Edition, total 3400 pages! The books are so large that we had to put several chapters from each on the accompanying CDs. This creates tremendous challenges and opportunities for us as authors, for our publisher—Prentice Hall, for instructors, for students and for professionals. We hope you enjoy the results of these challenges as much as we have enjoyed the process of tackling them.

Features of Advanced Java 2 Platform How to Program

This book contains many features including:

• **Full-Color Presentation.** This book is in full color to enable readers to see sample outputs as they would appear on a color monitor. Also, we now syntax color all the Java code, as do many of today’s Java integrated development environments and code editors. Our syntax-coloring conventions are as follows:
• “Code Washing.” This is our own term for the process we use to format the programs in the book with a carefully commented, open layout. The code is in full color and grouped into small, well-documented pieces. This greatly improves code readability—an especially important goal for us given that this book contains almost 40,000 lines of code.

• Advanced Graphical User Interface (GUI) Design. Starting with Chapter 2, we use advanced Java Swing features to create real-world Java components, including a Web-browser application with a multiple-document interface. In Chapter 3, we introduce the Model-View-Controller (MVC) architecture and its implementation in the Swing API. In Chapters 4 and 5, we create 2D graphics and 3D worlds. The Java 2D Drawing Application with Design Patterns Case Study in Chapter 5 presents a complex drawing program with which the user can create shapes in various colors and gradients. We are also pleased to add Java 3D coverage. One of the book’s adopters said these chapters were ideal for a course in advanced GUI programming. (We wanted to include multimedia programming with the Java Media Framework, but instead we decided to include this material in the companion book, Java How to Program, Fourth Edition.)

• Enterprise Java and Our Enterprise Java Case Study. Developers use Java for building “heavy-duty” enterprise applications. Chapters 7–11, 14–16 and 21 explore the necessary components for implementing enterprise solutions—including security, database manipulation, servlets, JavaServer Pages, distributed transactions, message-oriented middleware and application servers. In Chapter 7, Security, we discuss secure communications and secure programming. Chapters 17–20 showcase an Enterprise Java Case Study that integrates many technologies, such as Enterprise JavaBeans, servlets, RMI-IIOP, XML, XSLT, XHTML, (and for wireless application development) WML and cHTML—into an online-bookstore application. The Deitel Bookstore demonstrates how to use the MVC architecture introduced in Chapter 3 to build enterprise applications. This bookstore uses technologies to provide support for almost any type of client, including cell phones, mobile devices and Web browsers. In this world of networks and wireless networks, business information must be delivered securely and reliably to the intended recipients.

• Distributed Systems. Enterprise applications are usually so complex that they run more efficiently when program components are distributed among different machines in organizations’ networks. This book introduces several technologies for building distributed systems—Remote Method Invocation (RMI), Jini, JavaSpaces, Java Management Extensions (JMX), Jiro and Common Object Request Broker Architecture (CORBA). CORBA, controlled by the Object Management Group (OMG), is a mature distributed computing technology for integrating distributed components written in many disparate languages. Java was originally intended for networks of programmable devices—Jini assumes that technology role
now. JMX and Jiro are technologies specifically for network management (LANs, WANs, intranets, the Internet, extranets, etc.).

- **Java 2 Micro Edition (J2ME) and Wireless Applications.** It is estimated that by 2003, more people worldwide will access the Internet through wireless devices than through desktop computers. The Java platform for wireless devices with limited capabilities such as cell phones and personal digital assistants is Java 2 Micro Edition (J2ME). Chapter 12, Wireless Java-Based Applications Development and J2ME, contains a case study that sends content from a centralized data store to several wireless clients, including a J2ME client.

- **Web Services.** Web services are applications that expose public interfaces usable by other applications over the Web. The area of Web services builds on existing protocols, such as HTTP, and communicate with XML-based messages. Directory services enable clients to perform lookups to discover available Web services. The Simple Object Access Protocol (SOAP) uses XML to provide communication in many Web services. Many of the technologies in this book can be used to build Web services.

- **Employing Design Patterns.** The book’s largest case studies—such as the Java 2D drawing program in Chapter 5, the three-tier servlet and JavaServer Pages case study in Chapter 11, the three-tier wireless application in Chapter 12 and the Deitel Bookstore Enterprise Case Study in Chapters 17–20—each contain thousands of lines of code. Larger systems, such as automated teller machines or air-traffic control systems, can contain hundreds of thousands, or even millions, of lines of code. Effective design is crucial to the proper construction of such complex systems. Over the past decade, the software engineering industry has made significant progress in the field of *design patterns*—proven architectures for constructing flexible and maintainable object-oriented software.¹ Using design patterns can substantially reduce the complexity of the design process. We used many design patterns when building the software in this book. Chapter 1 introduces design patterns, discusses why they are useful and lists those design patterns we use throughout this book.

- **XML.** XML (Extensible Markup Language) use is exploding in the software-development industry and we use it pervasively throughout the text. As a platform-independent syntax for creating markup languages, XML’s data portability integrates well with Java’s portable applications and services. If you do not know XML, Appendices A–D of this book provide an introduction to XML. Appendices A and B introduce XML basics and DTDs, which define standard XML document structures. Appendix C introduces the Document Object Model (DOM) API for manipulating XML documents. Appendix D covers XSLT (Extensible Stylesheet Language Transformations—an XML vocabulary for transforming XML documents into other text-based documents.

- **Peer-to-Peer Applications.** Peer-to-peer (P2P) applications—such as instant messaging and document-sharing programs—have become extremely popular. Chap-

---
¹ Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software.* (Massachusetts: Addison-Wesley, 1995).
Chapter 28, Peer-to-Peer Applications and JXTA, introduces this architecture, in which each node performs both client and server duties. JXTA (short for the term “Juxtapose”), defines protocols for implementing peer-to-peer applications. This chapter includes two P2P application case studies—one written with Jini and RMI and the other written in multicast sockets and RMI. Both implement a P2P instant messaging application. We wanted a capstone example for Jini and decided this chapter should have it. The first case study is somewhat centralized—and therefore not a “true” P2P application (some developers think that Jini has too much overhead for a peer-to-peer application). We developed the second to demonstrate a lightweight, decentralized implementation.

- Appendix H, Career Opportunities. This appendix introduces career services on the Internet. We explore online career services from both the employer’s and employee’s perspectives. We suggest Web sites at which you can submit applications, search for jobs and review applicants (if you are interested in hiring someone). We also review services that build recruiting pages directly into e-businesses. One of our reviewers told us that he had just gone through a job search largely using the Internet and this chapter would have expanded his search dramatically.

- Appendix I, Unicode. This appendix overviews the Unicode Standard. As computer systems evolved worldwide, computer vendors developed numeric representations of character sets and special symbols for the local languages spoken in different countries. In some cases, different representations were developed for the same languages. Such disparate character sets made communication between computer systems difficult. Java supports the Unicode Standard (maintained by a non-profit organization called the Unicode Consortium), which defines a single character set with unique numeric values for characters and special symbols in most spoken languages. This appendix discusses the Unicode Standard, overviews the Unicode Consortium Web site (unicode.org) and shows a Java example that displays “Welcome” in many different languages.

- Bibliography and Resources. Chapters in this book contain bibliographies when appropriate and URLs that offer additional information about the technologies. We did this so those readers who would like to study a topic further could begin with the resources we found helpful when developing this book.

Some Notes to Instructors

A World of Object Orientation

When we wrote the first edition of Java How to Program, universities were still emphasizing procedural programming in languages like Pascal and C. The leading-edge courses were using object-oriented C++, but these courses were generally mixing a substantial amount of procedural programming with object-oriented programming—something that C++ lets you do, but Java does not. By the third edition of Java How to Program, many universities were switching from C++ to Java in their introductory curricula, and instructors were emphasizing a pure object-oriented programming approach. In parallel with this activity, the software engineering community was standardizing its approach to modeling ob-
ject-oriented systems with the UML, and the design-patterns movement was taking shape. This book takes a 100% object-oriented approach and emphasizes Java design patterns and adherence to Java idiom.

The prerequisite for this book is *Java How to Program, Fourth Edition* (or equivalent Java knowledge), which provides a solid foundation in Java programming. *Java How to Program, Fourth Edition* includes the following chapters and appendices, for a more detailed Table of Contents, visit [www.deitel.com](http://www.deitel.com): Introduction to Computers, the Internet and the Web; Introduction to Java Applications; Introduction to Java Applets; Control Structures: Part 1; Control Structures: Part 2; Methods; Arrays; Object-Based Programming; Object-Oriented Programming; Strings and Characters; Graphics and Java 2D; Graphical User Interface Components: Part 1; Graphical User Interface Components: Part 2; Exception Handling; Multithreading; Files and Streams; Networking; Multimedia: Images, Animation, Audio and Video; Data Structures; Java Utilities Package and Bit Manipulation; Collections; Java Media Framework and Java Sound; Java Demos; Java Resources; Operator Precedence Chart; ASCII Character Set; Number Systems; Creating HTML Documentation with javadoc; Elevator Events and Listener Interfaces; Elevator Model; Elevator View; Career Opportunities; Unicode; Bibliography.

**Students Like Java**

Students are highly motivated by the fact that they are learning a leading-edge language (Java) and a leading-edge programming paradigm (object-oriented programming) for building entire systems. Java immediately gives them an advantage when they head into a world in which the Internet and the World Wide Web have a massive prominence and corporations need enterprise systems programmers. Students quickly discover that they can do great things with Java, so they are willing to put in the extra effort. Java helps programmers unleash their creativity. We see this in the Java and advanced Java courses Deitel & Associates, Inc. teaches.

**Focus of the Book**

Our goal was clear—produce an advanced Java textbook for higher-level university courses in computer programming for students with intermediate-level Java programming experience, and offer the depth and the rigorous treatment of theory and practice demanded by professionals. To meet these goals, we produced a book that challenges Java programmers. We present clear examples of advanced topics and often overlooked topics. We adhere to Java idiom and follow sophisticated coding style and practices (i.e., not just the code formatting, but the idiomatic use of Java API’s, constructs and technologies). This book presents substantial Java applications that readers can use to start working with these technologies immediately.

**Evolution of Advanced Java 2 Platform How to Program**

*Advanced Java 2 Platform How to Program* was finished fresh on the heels of *Java How to Program, Fourth Edition*. Hundreds of thousands of university students and professionals worldwide have learned Java from our texts. Upon publication in September 2001, *Advanced Java 2 Platform How to Program* will be used in universities, corporations and government organizations worldwide. Deitel & Associates, Inc. taught Java courses internationally to thousands of students as we were writing the various editions of *Java How to
Preface XXVII

Program and Advanced Java 2 Platform How to Program. We carefully monitored the effec-
tiveness of material and tuned the books accordingly.

Conceptualization of Java
We believe in Java. Its conceptualization by Sun Microsystems, the creator of Java, was
brilliant. Sun based the new language on C and C++, two of the world’s most widely used
implementation languages. This immediately gave Java a huge pool of highly skilled pro-
grammers who were implementing most of the world’s new operating systems, communica-
tions systems, database systems, personal-computer applications and systems software.
Sun removed the more complex and error-prone C/C++ features (such as explicit pointers,
operator overloading and multiple inheritance, among others). They kept the language con-
cise by removing special-purpose features used by only small segments of the program-
ming community. They made the language truly portable for implementing Internet-based
and Web-based applications, and they included features developers need such as strings,
graphics, GUI components, exception handling, multithreading, multimedia (audio, imag-
es, animation and video), prepackaged data structures, file processing, database processing,
Internet and Web-based client/server networking, distributed computing and enterprise
computing. Then they made the language available at no charge to millions of potential
programmers worldwide.

2.5 Million Java Developers
Java was promoted in 1995 as a means of adding “dynamic content” to Web pages. Instead
of Web pages with only text and static graphics, Web pages could now “come alive” with au-
dios, videos, animations, interactivity—and soon, 3D imaging. But we saw much more in
Java than this. Java’s features are precisely what businesses and organizations need to meet
today’s information-processing requirements. So we immediately viewed Java as having the
potential to become one of the world’s key general-purpose programming languages. In fact,
Java has revolutionized software development with multimedia-intensive, platform-indepen-
dent, object-oriented code for conventional, Internet-, Intranet- and Extranet-based applica-
tions and applets. Java now has 2.5 million developers worldwide—a stunning
accomplishment when considering that it has been available publicly for only six years. No
other programming language has ever acquired such a large developer base so quickly.

Teaching Approach
Advanced Java 2 Platform How to Program, First Edition contains a rich collection of ex-
amples, exercises and projects drawn from many fields to provide readers with a chance to
solve interesting real-world problems. The book concentrates on the principles of good
software engineering and stresses program clarity, especially important when creating sub-
stantial programs like those covered in this book. We avoid arcane terminology and syntax
specifications in favor of teaching by example. Our code examples have been tested on
popular Java platforms. We are educators who teach edge-of-the-practice topics in industry
classrooms worldwide. The text emphasizes good pedagogy.

Learning Java via the live-code™ Approach
The book is loaded with live-code™ examples. This is how we teach and write about pro-
gramming, and is the focus of each of our multimedia Cyber Classrooms and Web-based
training courses. We present each new concept in the context of a complete, working Java program, immediately followed by screen captures that show the program’s output. We call this style of teaching and writing our **live-code™ approach**. *We use the language to teach the language.* Reading these programs (almost 40,000 lines of code) is much like entering and running them on a computer.

**Java Programming from Chapter Two**

*Advanced Java 2 Platform How to Program* “jumps right in” with substantial programs right from Chapter 2. This is the beginning of an aggressive pace that challenges readers with graphical, multithreaded, database-intensive, network-based programming. Throughout the book, readers learn by implementing impressive projects.

**World Wide Web Access**

All the code for *Advanced Java 2 Platform How to Program* is on the CD that accompanies this book. The code also is available at the following Web sites:

- [www.deitel.com](http://www.deitel.com)
- [www.prenhall.com/deitel](http://www.prenhall.com/deitel)

**Objectives**

Each chapter begins with *Objectives* that inform the reader what to expect and provides an opportunity, after reading the chapter, to determine if the reader has met these objectives. It is a confidence builder and a source of positive reinforcement.

**Quotations**

The learning objectives are followed by quotations. Some are humorous, some are philosophical and some offer interesting insights. Our readers enjoy relating the quotations to the chapter material. The quotations are worth a “second look” after you read each chapter.

**Outline**

The chapter outline helps the reader approach the material in top-down fashion. This, too, helps students anticipate what is to come and set a comfortable and effective learning pace.

**Almost 40,000 Lines of Code in 126 Example Programs (with Program Outputs)**

We present Java features in the context of complete, working Java programs. The programs in this book are substantial, with hundreds to thousands of lines of code (e.g., 10,000 lines of code for the bookstore case study example). Students should use the program code from the CD that accompanies the book and run each program while studying that program in the text.

**841 Illustrations/Figures**

Many of the figures are code examples, but this book still offers many charts, line drawings and program outputs. For example, Chapter 4 and 5, Graphics Programming with Java 2D and Java 3D, provides stunning graphics, and the architectural overview of the Enterprise Java case study in Chapter 17 is impressive.
235 Programming Tips
We have included programming tips to help students focus on important aspects of program development. We highlight numerous tips in the form of Good Programming Practices, Common Programming Errors, Testing and Debugging Tips, Performance Tips, Portability Tips, Software Engineering Observations and Look-and-Feel Observations. These tips and practices represent the best we have gleaned from decades of programming and teaching experience. One of our students—a mathematics major—told us that she feels this approach is like the highlighting of axioms, theorems and corollaries in mathematics books; it provides a basis on which to build good software.

Good Programming Practices
We highlight Good Programming Practices techniques for writing programs that are clearer, more understandable, more debuggable and more maintainable.

Common Programming Errors
Focusing on these Common Programming Errors helps readers avoid making the same errors.

Testing and Debugging Tips
When we first designed this “tip type,” we thought we would use it strictly to tell people how to test and debug Java programs. In fact, many of the tips describe aspects of Java that reduce the likelihood of “bugs” and thus simplify the testing and debugging process.

Performance Tips
We have included 13 Performance Tips that highlight opportunities for improving program performance—making programs run faster or minimizing the amount of memory that they occupy.

Portability Tips
One of Java’s “claims to fame” is “universal” portability, so some programmers assume that if they implement an application in Java, the application will automatically be “perfectly” portable across all Java platforms. Unfortunately, this is not always the case. We include Portability Tips to help readers write portable code and to provide insights on how Java achieves its high degree of portability.

Software Engineering Observations
The object-oriented programming paradigm requires a complete rethinking about the way we build software systems. Java is an effective language for performing good software engineering. The Software Engineering Observations highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.

Look-and-Feel Observations
We provide Look-and-Feel Observations to highlight graphical user interface conventions. These observations help readers design their own graphical user interfaces in conformance with industry norms.

Summary (949 Summary bullets)
Each chapter ends with additional pedagogical devices. We present a thorough, bullet-list-style summary of the chapter. On average, there are 26 summary bullets per chapter. This helps the readers review and reinforce key concepts.
Terminology (1904 Terms)
We include in a Terminology section an alphabetized list of the important terms defined in the chapter—again, further reinforcement. On average, there are 51 terms per chapter.

394 Self-Review Exercises and Answers (Count Includes Separate Parts)
Self-review exercises and answers are included for self-study. These reinforce the knowledge the reader gained from the chapter.

189 Exercises (Count Includes Separate Parts)
Each chapter concludes with a set of exercises. The exercises cover many areas. This enables instructors to tailor their courses to the unique needs of their audiences and to vary course assignments each semester. Instructors can use these exercises to form homework assignments, quizzes and examinations. The solutions for most of the exercises are included on the Instructor’s Manual CD that is available only to instructors through their Prentice-Hall representatives. [NOTE: Please do not write to us requesting the instructor’s manual. Distribution of this publication is strictly limited to college professors teaching from the book. Instructors may obtain the Instructor’s manual only from their Prentice Hall representatives. We regret that we cannot provide the solutions to professionals.] Solutions to approximately half of the exercises are included on the Advanced Java 2 Platform Multimedia Cyber Classroom CD, which also is part of The Complete Advanced Java 2 Platform Training Course. For ordering instructions, please see the last few pages of this book or visit www.deitel.com.

Approximately 3,080 Index Entries (with approximately 4648 Page References)
This book includes an extensive index. This helps the reader find any term or concept by keyword. The index is useful to developers who use the book as a reference. The terms in the Terminology sections generally appear in the index (along with many more index items from each chapter).

“Double Indexing” of Java live-code™ Examples and Exercises
Advanced Java 2 Platform How to Program has 126 live-code™ examples and 189 exercises (including parts). Many exercises are challenging problems or projects that require substantial effort. We have “double indexed” the live-code™ examples. For every Java source-code program in the book, we took the file name with the .java extension, such as WebBrowser.java and indexed it both alphabetically (in this case under “W”) and as a subindex item under “Examples.” This makes it easier to find examples using particular features.

Software Included with Advanced Java 2 Platform How to Program
There are a number of for-sale Java products available. However, you do not need them to get started with Java. We wrote Advanced Java 2 Platform How to Program using the Java 2 Software Development Kit (J2SDK) Standard Edition Version 1.3.1 for Windows and Linux (Intel x86) and other software programs that we include on the CD that accompanies this book. For your convenience, Sun’s J2SDK also can be downloaded from the Sun Microsystems Java Web site java.sun.com/j2se. We include some of the most popular
server software so you can set up and run live systems. This software includes *BEA WebLogic Server™, Version 6.0 (Windows/Linux) with Service Pack 2, 30-Day Trial, Enterprise Edition, 6.0, Testdrive; IBM® WebSphere® Application Server, Advanced Single Server Edition, Version 4.0 for Windows NT® and Windows® 2000 Evaluation Copy, and Apache Tomcat 3.2.3 from the Apache Software Foundation*. We also include Informix Software’s *Cloudscape 3.6.4* database software. With Sun’s cooperation, we also were able to include on the CD a powerful Java integrated development environment (IDE)—Sun Microsystem’s *Forte for Java Community Edition*. *Forte* is a professional IDE written in Java that includes a graphical user interface designer, code editor, compiler, visual debugger and more. J2SDK 1.3.1 must be installed before installing *Forte*. If you have any questions about using this software, please read the introductory *Forte* documentation on the CD. We will provide additional information on our Web site www.deitel.com.

The CD also contains the book’s examples and a Web page with links to the Deitel & Associates, Inc. Web site (www.deitel.com), the Prentice Hall Web site (www.prenhall.com/deitel) and the many Web sites listed at the end of each chapter. If you have access to the Internet, this Web page can be loaded into your Web browser to give you quick access to all the resources. Finally, because we wrote much more than we originally intended, a number of chapters and appendices have been off-loaded to the CD.

**Ancillary Package for Advanced Java 2 Platform How to Program**

*Advanced Java 2 Platform How to Program* has extensive ancillary materials for instructors teaching from the book. The Instructor’s Manual CD contains solutions to the vast majority of the end-of-chapter exercises. We also provide PowerPoint® slides containing all the code and figures in the text. You are free to customize these slides to meet your own classroom needs. Prentice Hall provides a *Companion Web Site* (www.prenhall.com/deitel) that includes resources for instructors and students. For instructors, the Web site has a *Syllabus Manager* for course planning, links to the PowerPoint slides and reference materials from the appendices of the book (such as the character sets and Web resources). For students, the Web site provides chapter objectives, true/false exercises with instant feedback, chapter highlights and reference materials. *[NOTE: Please do not write to us requesting the instructor’s manual. Distribution of this publication is strictly limited to college professors teaching from the book. Instructors may obtain the solutions manual only from their regular Prentice Hall representatives. We regret that we cannot provide the solutions to professionals.]*

**Advanced Java 2 Platform Multimedia Cyber Classroom (CD and Web-Based Training Versions) and The Complete Advanced Java 2 Platform Training Course**

We have prepared an interactive, CD-based, software version of *Advanced Java 2 Platform How to Program*, called the *Advanced Java 2 Platform Multimedia Cyber Classroom*. It is loaded with features for learning and reference. The *Cyber Classroom* is wrapped with the textbook at a discount in *The Complete Advanced Java 2 Platform Training Course*. If you already have the book and would like to purchase the *Advanced Java 2 Platform Multimedia Cyber Classroom* (ISBN: 0-13-091276-x) separately, please visit www.infor-
mit.com/cyberclassrooms. All Deitel Cyber Classrooms are generally available in CD and Web-based training formats.

The CD has an introduction with the authors overviewing the Cyber Classroom’s features. Many of the live-code™ examples in the textbook truly “come alive” in the Cyber Classroom. If you are viewing a program and want to execute it, you click the lightning bolt icon and the program will run. You will immediately see—and hear for the audio-based multimedia programs—the program’s outputs. If you want to modify a program and see and hear the effects of your changes, simply click the floppy-disk icon that causes the source code to be “lifted off” the CD and “dropped into” one of your own directories so you can edit the text, recompile the program and try out your new version. Click the audio icon and one of the authors will talk about the program and “walk you through” the code.

The Cyber Classroom also provides navigational aids including extensive hyperlinking. The Cyber Classroom is browser based, so it remembers recent sections you have visited and allows you to move forward or backward among these sections. The thousands of index entries are hyperlinked to their text occurrences. You can search for a term using the “find” feature and the Cyber Classroom locates its occurrences throughout the text. The Table of Contents entries are “hot”—so clicking a chapter name takes you to that chapter.

Students tell us that they particularly like the fact that solutions to about half the exercises in the book are included with the Cyber Classroom. Studying and running these extra programs is a great way for students to enhance their learning experience.

Students and professional users of our Cyber Classrooms tell us they like the interactivity and that the Cyber Classroom is an effective reference because of the extensive hyperlinking and other navigational features. We received an email from a person who said that he lives “in the boonies” and cannot take a live course at a university, so the Cyber Classroom was the solution to his educational needs.

Professors tell us that their students enjoy using the Cyber Classroom, spend more time on the course and master more of the material than in textbook-only courses. We have published (and will be publishing) many other Cyber Classroom and Complete Training Course products. For a complete list of the available and forthcoming Cyber Classrooms and Complete Training Courses, see the Deitel™ Series page at the beginning of this book or the product listing and ordering information at the end of this book. You can also visit www.deitel.com or www.prenhall.com/deitel for more information.

Acknowledgments

One of the great pleasures of writing a textbook is acknowledging the efforts of the many people whose names may not appear on the cover, but whose hard work, cooperation, friendship and understanding were crucial to the production of the book.

Several people at Deitel & Associates, Inc. devoted long hours to this project. We would like to acknowledge the efforts of our full-time Deitel & Associates, Inc. colleagues Jonathan Gadzik, Tem Nieto, Su Zhang, Kyle Lomeli, Matthew Kowalewski, Rashmi Jayaparakash, Kate Steinbuhler, Abbey Deitel and Betsy DuWaldt.

- Jonathan Gadzik, a graduate of the Columbia University School of Engineering and Applied Science (BS in Computer Science) co-authored Chapter 1, Introduction, and Chapter 12, Java-Based Wireless Applications Development and J2ME, and contributed to Chapter 4 and the design patterns material throughout the book. He also reviewed Chapter 28, Peer-to-Peer Applications.
• Tem Nieto, a graduate of the Massachusetts Institute of Technology, is Director of Product Development at Deitel & Associates. Tem teaches XML, Java, Internet and Web, C, C++ and Visual Basic seminars and works with us on textbook writing, course development and multimedia-authoring efforts. He is co-author with us of *Internet & World Wide Web How to Program (Second Edition)*, *XML How to Program*, *Perl How to Program* and *Visual Basic 6 How to Program*. In *Advanced Java 2 Platform How to Program, First Edition* Tem updated Chapters 5, 6, 8 and 12 of *XML How to Program* for inclusion as Appendices A–D—Creating Markup with XML, XML Document Type Definitions, XML Document Object Model (DOM) and XSL (Extensible Stylesheet Language Transformations)—respectively.

• Su Zhang, a graduate of McGill University with a Masters in Computer Science, co-authored Chapters 22, 23, 24 and 25—Jini, JavaSpaces, Jiro and JMX, respectively.

• Kyle Lomeli, a graduate of Oberlin College in Computer Science co-authored Chapters 24 and 25 (JMX and Jiro). He contributed to Chapter 3, MVC; Chapter 7, Security; Chapter 13, RMI and Chapter 23, JavaSpaces, and he reviewed Chapter 12.

• Matthew Kowalewski, a graduate of Bentley College with a major in Accounting Information Systems and Director of Wireless Development at Deitel & Associates, Inc., contributed to Chapter 12.

• Rashmi Jayaprakash, a graduate of Boston University with a major in Computer Science, co-authored Appendix I, Unicode.

• Kate Steinbuhler, a graduate of Boston College with majors in English and Communications, co-authored Appendix H, Career Opportunities, and managed the permissions process.


• Betsy DuWaldt, a graduate of Metropolitan State College of Denver with a degree in Technical Communications (Writing and Editing Emphasis) and a minor in Computer Information Systems, is Editorial Director at Deitel & Associates, Inc. She co-authored the Preface, helped prepare the manuscript for publication and edited the index.

We would like to thank the participants in our Deitel & Associates, Inc. College Internship Program.²

• Chris Henson, a Masters student at Brandeis University (Computer Science), co-authored Chapter 6, JavaBeans Component Model, and Chapter 29, Web Services. He contributed to the accessibility section of Chapter 2, reviewed Chapters 21

---

² The *Deitel & Associates, Inc. College Internship Program* offers a limited number of salaried positions to Boston-area college students majoring in Computer Science, Information Technology, Marketing, English or Technical Writing. Students work at our corporate headquarters in Sudbury, Massachusetts full-time in the summers and part-time during the academic year. Full-time positions are available to college graduates. For more information about this competitive program, please contact Abbey Deitel at deitel@deitel.com and check our Web site, www.deitel.com.
and 22, 23, 25 and Appendix I and applied technical reviews to Chapters 2, 6, 8, 14, 15 and 29.

- Audrey Lee, a Senior at Wellesley College in Computer Science and Mathematics, co-authored Chapter 16, Java Message Service and contributed to Chapters 7, 13, 18 and Appendices F and I.

- Jeffrey Hamm, a Sophomore in Computer Science at Northeastern University, co-authored Chapter 21, Appendix E and Appendix G, Java Native Interface (JNI).

- Varun Ganapathi, a Sophomore in Computer Science and Electrical Engineering at Cornell University, co-authored Chapter 28, contributed to Chapter 12 and implemented the i-mode and WML clients in the Chapter 18 case study.

- Sasha Devore, a graduate of Massachusetts Institute of Technology in Electrical Engineering and Electrical Science, 2001, co-authored Chapter 4, Graphics Programming with Java 2D and Java 3D.

- A. James O’Leary, a sophomore in Computer Science and Psychology at Rensselaer Polytechnic Institute, co-authored Chapter 7, Security.

- Susan Warren, a Junior in Computer Science at Brown University, worked on the Instructor’s Manual and ancillary materials for Chapters 9 and 10.

- Eugene Izumo, a Sophomore in Computer Science at Brown University, worked on the Instructor’s Manual and ancillary materials for Chapters 9 and 10.

- Vincent He, a Senior in Management and Computer Science at Boston College, worked on the Instructor’s Manual for Chapter 8.

- Christina Carney, a Senior in Psychology and Business at Framingham State College helped prepare the Preface and the bibliography for several chapters.

- Amy Gips, a Sophomore in Marketing and Finance at Boston College, co-authored Appendix F, Java Community Process, and researched URLs for several chapters. Amy also researched the quotes for the entire book and helped prepare the Preface.

- Fabian Morgan (a Summer 2000 intern from MIT) wrote the initial versions of the examples for Chapters 5, 8, 14, 15 and the Enterprise Java case study in Chapters 17–20.

- Josh Gould (a Summer 2000 intern from Clark University) worked on Chapters 9 and 10.

We also would like to thank two business colleagues who contributed to the book.

- Carlos Valcarcel co-authored Chapters 26 and 27. Carlos is an independent OO/Java/CORBA architect with EinTech, Inc., in New York. Carlos has been working with Java since November 1995 and CORBA since mid-1996. His clients range from investment banks and insurance companies to software vendors. Please feel free to send questions and comments to Carlos at carlos@eintech.com. Carlos would like to thank his wife Becky and daughter Lindley for their patience and understanding during the writing of these two chapters.

"If there is a bright center to the universe, the two of you are it.”
Kelby Zorgdrager served as a technical consultant on Chapter 22, Jini, Chapter 23, JavaSpaces, Chapter 24, JMX and Chapter 25, Jiro. He has been working with Java since its beginning stages of JDK 1.0. Over the past 5 years, Kelby has worked for Sun Microsystems as a Java Instructor where he developed course materials and presented to over 3500 students worldwide. During Kelby's last year at Sun, he worked as a Software Engineer on the development of the Jiro Technology. Kelby has spoken at internationally recognized industry conferences, including JavaOne. Currently, Kelby is working as the Director of Architecture for eCarCredit.com, where he uses Java to create cutting-edge technological solutions for the Auto Finance Industry. In Kelby's spare time, he provides independent consulting services, and enjoys spending time with his wife Beth, daughter Aubreigh, and Winston the St. Bernard. Kelby can be reached at advanced_java@zorgdrager.org.

We also would like to thank those people who helped us obtain commercial application server software for the CD that accompanies this book and those people who helped us complete the deployment instructions for our Deitel Bookstore case study on the three most popular application servers. Our thanks to Katherine Barnhisel of BEA Systems; Sheila Richardson, John Botsford, Jason McGee and Kevin Vaughan of IBM; and Holly Sharp, Heather Sutherland, Sharada Achanta, Patrick Dorsey and Deepak Balakrishna of iPlanet.

We are fortunate to have been able to work on this project with the talented and dedicated team of publishing professionals at Prentice Hall. We especially appreciate the extraordinary efforts of our computer science editor, Petra Recter and her boss—our mentor in publishing—Marcia Horton, Editor-in-Chief of Prentice-Hall’s Engineering and Computer Science Division. Vince O’Brien and Camille Trentacoste did a marvelous job handling production.

The Advanced Java 2 Platform Multimedia Cyber Classroom was developed in parallel with Advanced Java 2 Platform How to Program. We sincerely appreciate the “new media” insight, savvy and technical expertise of our e-media editor-in-chief, mentor and friend Mark Taub. He and our e-media editor, Karen McLean, did a remarkable job bringing the Advanced Java 2 Platform Multimedia Cyber Classroom to publication under a tight schedule. Michael Ruel did a marvelous job as Cyber Classroom project manager.

We owe special thanks to the creativity of Tamara Newnam Cavallo (smart_art@earthlink.net) who did the art work for our programming tips icons and the cover. She created the delightful bug creature who shares with you the book’s programming tips. Barbara Deitel contributed the bugs’ names on the front cover.

We sincerely appreciate the efforts of our reviewers:

Jeff Allen (Sun Microsystems)
Dibyendu Baksi (Sun Microsystems)
Tim Boudreau (Sun Microsystems)
Paul Byrne (Sun Microsystems)
Onno Kluyt (Sun Microsystems)
Peter Korn (Sun Microsystems)
Petr Kozel (Sun Microsystems)
Jon Nyquist (Sun Microsystems)
Tomas Pavek (Sun Microsystems)
Martin Ryzl (Sun Microsystems)
Davanum Srinivas (JNI-FAQ Manager, Sun Microsystems)
Brandon Taylor (Sun Microsystems)

Vicki Allan (Utah State University)
Javaid Aslam (Analyst/Application Developer, Tektronix)
Henry Balen (CORBA author)
Kathy Barshatzky (Javakathy.com)
Don Benish (Ben-Cam Intermedia)
Keith Bigelow (Lutris)
Darrin Bishop (Levi, Ray and Shoup, Inc.)
Ron Braithwaite (Nutriware)
Carl Burnham (Southpoint)
John Conley (DeVry Institute)
Charles Costarella (Antelope Valley College)
Jonathan Earl (Technical Training Consultants)
Jesse Glick (NetBeans)
Ken Gilmore (Amdocs, Inc.)
Jason Gordon (Verizon)
Christopher Green (Colorado Springs Technical Consultants)
Michele Guy (XOR)
Deborah Hooker (Mnemosyne Consulting)
Elizabeth Kallman (Los Alamos National Library)
Salvi Karuppaswamy (EDS)
Jodi Krochalis (Compuware)
Anthony Levensalor (Compuware)
Derek Lane (President of Gunslinger Software and Consulting, Inc.)
Rick Loek (Callidus Software)
Ashish Makhijani (Senior Analyst, Programmer)
Paul McLachlan (Compuware)
Randy Meyers (NetCom)
Paul Monday (Imation)
Steven Newton (Lead Programmer/Analyst, Standard Insurance Company)
Victor Peters (NextStepEducation)
Bryan Plaster (Valtech)
Brian Pontarelli (Consultant)
Srikanth Raju (Staff Engineer, Sun Microsystems)
Robin Rowe (MovieEditor.com)
Michael Schmaltz (Accenture)
Joshua Sharff (Joshua Sharff Associates)
Dan Shellman (Software Engineer)
Jon Siegel (OMG)
Uma Subbiah (Unigraphics)
Arun Taksali (jataayusoft)
Vadim Tkachenko (Sera Nova)
Kim Topley (Author of Core Java Foundation Classes and Core Swing: Advanced Programming, both published by Prentice Hall)
John Varghese (University of Rochester)
Xinju Wang (Emerald Solutions)
Karen Wieslewski (Titan Insurance)
Jesse Wilkins (Metalinear Media)
Under a tight time schedule, they scrutinized every aspect of the text and made countless suggestions for improving the accuracy and completeness of the presentation.

We would sincerely appreciate your comments, criticisms, corrections, and suggestions for improving the text. Please address all correspondence to:

deitel@deitel.com

We will respond immediately. Well, that’s it for now. Welcome to the exciting world of Java programming. We hope you enjoy this look at leading-edge computer applications development. Good luck!

Dr. Harvey M. Deitel
Paul J. Deitel
Sean E. Santry

About the Authors

Dr. Harvey M. Deitel, CEO of Deitel & Associates, Inc., has 40 years experience in the computing field including extensive industry and academic experience. He is one of the world’s leading computer science instructors and seminar presenters. Dr. Deitel earned B.S. and M.S. degrees from the Massachusetts Institute of Technology and a Ph.D. from Boston University. He has 20 years of college teaching experience including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc. with his son Paul J. Deitel. He is author or co-author of dozens of books and multimedia packages and is currently writing many more. With translations published in Japanese, Russian, Spanish, Italian, Basic Chinese, Traditional Chinese, Korean, French, Polish and Portuguese, the Deitel’s texts have earned international recognition. Dr. Deitel has delivered professional seminars internationally to major corporations, government organizations and various branches of the military.

Paul J. Deitel, Chief Technical Officer of Deitel & Associates, Inc., is a graduate of the Massachusetts Institute of Technology’s Sloan School of Management where he studied Information Technology. Through Deitel & Associates, Inc. he has delivered Internet and World Wide Web courses and programming language classes for industry clients including Sun Microsystems, EMC², IBM, BEA Systems, Visa International, Progress Software, Boeing, Fidelity, Hitachi, Cap Gemini, Compaq, Art Technology, White Sands Missile Range, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, Rogue Wave Software, Lucent Technologies, Computervision, Cambridge Technology Partners, Adra Systems, Entergy, CableData Systems, Banyan, Stratus, Concord Communications and many other organizations. He has lectured on Java and C++ for the Boston Chapter of the Association for Computing Machinery, and has taught satellite-based courses through a cooperative venture of Deitel & Associates, Inc., Prentice Hall and the Technology Education Network. He and his father, Dr. Harvey M. Deitel, are the world’s best-selling Computer Science textbook authors.

Sean E. Santry, Director of Software Development with Deitel & Associates, Inc., is a graduate of Boston College where he studied computer science and philosophy. At Boston College he performed original research on the application of metaphysical systems to object-oriented software design. Through Deitel & Associates, Inc. he has delivered advanced and introductory courses for industry clients including Sun Microsystems,
EMC², Dell, Compaq, Boeing and others. He has contributed to several Deitel publications, including *Java How to Program, Fourth Edition; XML How to Program; C++ How to Program, Third Edition; C How to Program, Third Edition; e-Business and e-Commerce How to Program* and *e-Business and e-Commerce for Managers*. Before joining Deitel & Associates, he developed e-business applications with a leading Boston-area consulting firm.

**About Deitel & Associates, Inc.**

Deitel & Associates, Inc. is an internationally recognized corporate training and content-creation organization specializing in Internet/World Wide Web software technology, e-business/e-commerce software technology and computer programming languages education. Deitel & Associates, Inc. is a member of the World Wide Web Consortium. The company provides courses on Internet and World Wide Web programming, object technology and major programming languages. The founders of Deitel & Associates, Inc. are Dr. Harvey M. Deitel and Paul J. Deitel. The company’s clients include many of the world’s largest computer companies, government agencies, branches of the military and business organizations. Through its publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming textbooks, professional books, interactive CD-ROM-based multimedia *Cyber Classrooms, Complete Training Courses* and Web-based training courses. Deitel & Associates, Inc. and the authors can be reached via e-mail at

*deitel@deitel.com*

To learn more about Deitel & Associates, Inc., its publications and its worldwide corporate on-site curriculum, see the last few pages of this book and visit:

*www.deitel.com*

Individuals wishing to purchase Deitel books, *Cyber Classrooms, Complete Training Courses* and Web-based training courses can do so through

*www.deitel.com*

Bulk orders by corporations and academic institutions should be placed directly with Prentice Hall. See the last few pages of this book for worldwide ordering details.

**The World Wide Web Consortium (W3C)**

Deitel & Associates, Inc. is a member of the *World Wide Web Consortium (W3C)*. The W3C was founded in 1994 “to develop common protocols for the evolution of the World Wide Web.” As a W3C member, we hold a seat on the W3C Advisory Committee (our Advisory Committee representative is our Chief Technology Officer, Paul Deitel). Advisory Committee members help provide “strategic direction” to the W3C through meetings around the world. Member organizations also help develop standards recommendations for Web technologies (such as HTML, XML and many others) through participation in W3C activities and groups. Membership in the W3C is intended for companies and large organizations. For information on becoming a member of the W3C visit *www.w3.org/Consortium/Prospectus/Joining*.
Introduction

Objectives

• To understand the organization of the book.
• To understand various setup issues in deploying the book’s examples.
• To understand the elements of design patterns and how they are used throughout the book.
• To tour the book.

Before beginning, plan carefully.
Marcus Tullius Cicero

Things are always at their best in the beginning
Blaise Pascal

High thoughts must have high language.
Aristophanes

Our life is frittered away by detail ... Simplify, simplify
Henry Thoreau

Look with favor upon a bold beginning.
Virgil

I think I’m beginning to learn something about it.
Auguste Renoir
1.1 Introduction

Welcome to the world of advanced Java 2 Platform programming! We have worked hard to create what we hope will be an informative, entertaining and challenging learning experience for you.

The Java technologies you will learn are intended for developers and software engineers. *Advanced Java 2 Platform How to Program* presumes knowledge of either *Java How to Program: Fourth Edition* (ISBN: 0-13-034151-7) or *The Complete Java Training Course, Fourth Edition* (ISBN: 0-13-064931-7), which teach the fundamentals of Java and object-oriented programming. *Advanced Java 2 Platform How to Program* presents many advanced Java topics and introduces many new topics, using almost 40,000 lines of complete, working code and numerous illustrations to demonstrate the concepts. We integrate these technologies into substantial applications and enterprise systems that demonstrate how the pieces fit together. We call this our Live-Code™ approach.

We introduce technologies from the three Java editions—*Java 2 Standard Edition (J2SE)*, *Java 2 Enterprise Edition (J2EE)* and *Java 2 Micro Edition (J2ME)*. The beginning chapters of this book demonstrate several high-end concepts from J2SE (*Java How to Program, Fourth Edition* presents J2SE through the intermediate level). *Advanced Java 2 Platform How to Program* highlights many advanced features of J2EE, providing enterprise applications as examples. Finally, we introduce the exciting, leading-edge technologies of J2ME and wireless applications programming.
Object-oriented programming and design patterns are essential for building applications using the many technologies introduced in this book. These tools encourage modularity, allowing programmers to design classes and programs effectively. Design patterns in particular have proven critical to producing the substantial programs we present in this book.

Many of the book’s applications integrate the Extensible Markup Language (XML), the standard for creating markup languages that describe structured data in a platform-independent manner. Everything from common Web pages to complex order-tracking and business-to-business (B2B) systems can use XML. XML’s data portability complements the portability of programs built for the Java 2 Platform. XML’s capabilities for describing data enable systems built with disparate technologies to share data without concerns for binary compatibility, which is key to developing interoperable distributed systems in Java. We assume knowledge of XML and Java’s XML APIs. However, Appendices A–D also provide an introduction to XML and Java’s XML APIs for those of you who are not yet familiar with these topics. It is highly recommended that you read these appendices first, if you are not already familiar with XML.

As you read this book, you may want to refer to our Web site www.deitel.com for updates and additional information on the cutting-edge technologies you will be learning.

1.2 Architecture of the Book

There are several broad technology categories that comprise Advanced Java 2 Platform How to Program. Many of these technologies are inter-related. We begin with a discussion of these categories and an overview of the architecture of the book. The chapters can be grouped into several advanced topics—advanced GUI and graphics, distributed systems, Web services, Enterprise Java and XML technologies.

1.2.1 Advanced GUI, Graphics and JavaBeans

Chapters 2–6

Graphical user interfaces help users interact effectively and efficiently with applications. When creating substantial client applications, it is important to create simple and attractive user interfaces that enable users to work with your application in an intuitive and convenient manner. Java’s Swing API provides graphical user interface components common to many windowed applications and platforms. Java How to Program, Fourth Edition provides an introduction to GUI concepts with Swing. In Chapter 2 of Advanced Java 2 Platform How to Program, we introduce several more advanced Swing components and use them to create substantial applications such as a Web browser with a multiple-document interface. We also introduce Java’s capabilities for building applications for global deployment (through internationalization) and for building accessible applications for people with disabilities (using the Accessibility APIs).

A fundamental theme in Advanced Java 2 Platform How to Program is the importance of design patterns for building object-oriented systems. We use several design patterns when building the programming examples in this book. This chapter (Section 1.5) introduces design patterns, discusses why they are important and lists by chapter those design patterns we use in the book. Chapter 3 introduces the Model-View-Controller (MVC) archi-
tecture, which is based on several design patterns. This widely applicable architecture separates the presentation of data (e.g., a bar-chart showing bank-account information) from the underlying data representation (e.g., tables in a database) and the control logic for those data (e.g., event handlers for buttons and text fields in a user interface). In Chapter 3, we discuss the MVC architecture and its implementation in the Swing API. In later chapters, we revisit the MVC architecture and use it to build substantial Enterprise Java applications.

In Chapter 4, we present Java’s support for graphics. Java provides the Java 2D™ API for creating two-dimensional graphics and the Java 3D™ API for creating three-dimensional, virtual worlds. We introduce and demonstrate these APIs and provide examples including a three-dimensional game.

Chapter 5 contains a substantial case study—a Java 2D drawing application with design patterns—in which we present a complex drawing program as a capstone for the advanced GUI portion of the book. Using MVC and several other design patterns, and the capabilities of Java’s Swing components and Java 2D, our drawing application provides several types of shapes, various colors, gradients, image capabilities and more. Users can choose multiple views for a drawing, including a zoomed detail view.

The JavaBeans component model enables developers to “componentize” their applications, making those applications more flexible and the application components more reusable. We introduce JavaBeans (often called simply beans) in the context of an animation application in Chapter 6. JavaBeans allow programmers to create components for building applications; programmers called component assemblers then can assemble these components, along with existing components, to create applications, applets or even new beans. In fact, most of the GUI components presented in earlier chapters are JavaBeans.

1.2.2 Distributed Systems

Chapters 13, 22–28

When creating substantial applications, often it is more efficient, or even necessary, for concurrent tasks to be performed on different computers. Distributed systems technologies enable applications to execute across several computers on a network. For a distributed system to function correctly, application components executing on different computers throughout a network must be able to communicate with one another. Advanced Java 2 Platform How to Program presents several technologies for building distributed systems.

Chapter 13 introduces Remote Method Invocation (RMI), which allows Java objects located on different computers or executing in different virtual machines to interact as if they were on the same computer or in the same virtual machine. Each object invokes methods on the other objects and RMI handles the marshalling (i.e., collecting and packaging) of arguments and return values passed between remote objects. We present several RMI examples, including a distributed chat application.

Java also provides higher-level APIs for building distributed systems, including Jini and JavaSpaces. Jini (Chapter 22) enables devices or software programs on a LAN to interoperate without the need to install special device drivers, and with reduced administrative overhead. Jini provides true “plug-and-play” support for devices—just plug a printer into a network and that printer’s services become available to everyone on that network. JavaSpaces is a Jini service that provides a simple but powerful API for building distributed systems. We demonstrate JavaSpaces technology by building a distributed image processing application.
As networks grow in complexity and as companies depend on those networks more heavily for conducting business, network management grows in importance. The Java Management Extensions (JMX, Chapter 24) and Jiro (Chapter 25) are two complementary technologies for building distributed network management applications in Java.

In Chapters 26–27, we introduce CORBA—the Common Object Request Broker Architecture. CORBA allows programs written in various languages, with varying implementations running in disparate locations, to communicate with each other as easily as if they were in the same process address space. In these chapters, we introduce the fundamentals of CORBA and compare CORBA with other distributed-systems technologies, such as RMI. We also introduce RMI-IIOP, which enables RMI to interoperate with CORBA.

In Chapter 28, we discuss fundamental concepts of peer-to-peer (P2P) applications, where each application performs both client and server functions, thus distributing processing and information across many computers. We present two different implementations of a P2P instant-messaging application. The first implementation uses Jini technology and the second uses multicast sockets and RMI.

1.2.3 Web Services

Chapters 9–12, 29

The popularity of the Web and its importance for conducting business have exploded in recent years. The field of Web services is concerned with building services that enable information sharing, commerce and other interactions between businesses, between businesses and consumers, etc., using standard Web protocols. Web services have come about through an evolution of existing Web technologies, such as HTML forms, and enterprise technologies, such as messaging and Electronic Document Interchange (EDI) systems. Web services rely upon existing protocols and standards.

Chapter 9 introduces servlets. Servlets can generate documents dynamically (e.g., XHTML documents) to send to clients in response to requests for information. Chapter 10 introduces Java Server Pages (JSP), which also deliver dynamic content to clients. JSPs dynamically serve Web content by using scriptlets and JavaBeans components in the context of a document. These documents are translated into servlets by the JSP container—i.e., the server application responsible for handling requests for JSPs. Chapter 11 presents a case study that serves as a capstone to the technology presented in Chapters 9 and 10. The case study integrates JavaBeans, servlets, JSPs, XML and XSLT to create an online bookstore.

Several new technologies, such as the Wireless Application Protocol (WAP), Wireless Markup Language (WML), i-mode and Java 2 Micro Edition (J2ME) have emerged for use with wireless devices. Chapter 12 introduces these wireless technologies, and uses them to construct a three-tier application that uses servlets and XML to deliver content to several wireless devices.

Chapter 29 introduces Web services—applications that expose public interfaces usable by other applications over the Web. Web services are accessible through HTTP and other Web protocols, and communicate with XML-based messages. Directory services enable clients to perform lookups to discover available Web services. The Simple Object Access Protocol (SOAP) uses XML to provide communication in many Web services. SOAP allows applications to make remote procedure calls to a Web service’s public methods. In this chapter, we implement a weather service that provides local forecasts from the National Weather Service, using SOAP.
1.2.4 Enterprise Java

Chapters 7, 8, 14–16, 21
Java has become enormously popular for building enterprise applications. Sun originally conceived of Java as a programming language for building small programs embedded in Web pages; since its inception, Java has grown into an industrial strength, enterprise-development language. At the 2001 JavaOne conference, Sun Microsystems announced that over 96% of enterprise application servers support the Java 2 Enterprise Edition.

Security is a primary concern for Java applications of all types, including enterprise applications. In Chapter 7 we introduce the fundamentals of security, including cryptography, digital signatures, authentication, authorization and public-key infrastructure. We also introduce Java’s sandbox security model, the Java Cryptography Extensions (JCE), the Java Secure Sockets Extensions (JSSE) and the Java Authentication and Authorization Services (JAAS).

An integral part of powerful software applications is the storage, retrieval and display of data. Substantial amounts of data are organized and stored in databases. Programmers often need to interact with databases to update or retrieve information. Chapter 8 introduces Java Database Connectivity (JDBC) for manipulating databases. We present examples that interact with the Cloudscape database management system from Informix Software. Cloudscape is available for download at www.cloudscape.com.

Business logic forms the core functionality of an enterprise application. Business logic is responsible for implementing the complex business rules that businesses require for transaction and information processing. In Chapter 14, we introduce the Enterprise JavaBean (EJB) component model for building enterprise application business logic. In particular, we discuss session EJBs for business logic, and distributed transactions, which enable EJBs to work across multiple databases and still maintain data integrity. In Chapter 15, we present entity EJBs, which enable developers to build and object-based layer for accessing information in long-term storage, such as a database.

Enterprise applications require extensive services and support at runtime for accessing databases, enabling distributed transactions, maintaining performance, etc. Application servers provide a rich runtime environment for enterprise application components. In Chapter 21, we introduce the three most popular commercial application servers—BEA’s WebLogic, IBM’s WebSphere and the iPlanet Application Server. We also provide complete instructions for deploying an enterprise-application case study on BEA’s WebLogic and IBM’s WebSphere.

1.2.5 Enterprise Case Study

Chapters 17–20
Chapters 17–20 present a capstone application for the Enterprise Java topics presented in Advanced Java 2 Platform How to Program—an Enterprise Java case study that integrates many Java technologies into a substantial 10,000 lines of code online bookstore application. In this case study, we build the Deitel Bookstore enterprise application using Enterprise JavaBeans with container-managed persistence, servlets, RMI-IIOP, XML, XSLT, XHTML, WML and cHTML. A fundamental feature of this example is that the bookstore uses XML and XSLT to provide support for virtually any type of client, including standard
Web browsers and mobile devices, such as cell phones. The modular, extensible architecture enables developers to implement support for additional client types simply by providing appropriate XSLT documents that translate XML documents into content appropriate for those client types. The Deitel Bookstore case study also demonstrates the Model-View-Controller (MVC) architecture in the context of an Enterprise Java application.

1.2.6 XML

Appendices A–D

Many examples throughout Advanced Java 2 Platform How to Program use XML. As a platform-independent language for creating markup languages, XML integrates well with Java applications. Unlike HTML, with which Web designers use to format information for display, XML provides structure and semantics for application data, but it does not format data. Developers can create XML grammars that define the structure for data and make those data interoperable with other applications. The Java API for XML parsing (JAXP) provides the Java 2 Platform with a common API for manipulating XML parsers and XML data across platforms. The Document Object Model, Level 2 API (DOM) is backed by the World Wide Web Consortium (W3C) as a standard API for building and manipulating XML documents. Using this API, developers can leverage the cross-platform capabilities of Java and XML to build powerful distributed systems.

We introduce the basics of XML in Appendix A, Creating XML Markup. Appendix B introduces Document Type Definitions (DTDs) for defining standard document structures against which XML parsers can validate XML documents. DTDs are crucial for building XML documents that interoperate across many applications. Appendix C introduces the Document Object Model (DOM) API and its use in the Java API for XML Processing (JAXP). Appendix D introduces Extensible Stylesheet Language Transformations (XSLT), which is an XML grammar for transforming XML documents into other XML documents. We use XSLT in several examples to transform raw XML data into appropriate markup for Web clients, such as standard Web browsers and cell phones.

1.3 Tour of the Book

In this section, we include walkthroughs of each chapter and outline the many Java technologies discussed in Advanced Java 2 Platform How to Program. There will be terms in these sections that are unfamiliar to you—they will be defined in the chapters of the book. Many chapters end with an Internet and World Wide Web Resources section that provides a listing of Web sites you should visit to enhance your knowledge of the technologies discussed in that chapter. You may also want to visit the Web sites www.deitel.com and www.prenhall.com/deitel to keep informed of the latest information, book errata and additional teaching and learning resources.

Chapter 1—Introduction

This chapter overviews the technologies presented in Advanced Java 2 Platform How to Program and introduces the architecture of the book—advanced GUI and graphics, distributed systems, Web services, Enterprise Java and XML technologies. We include a tour of the book with a brief overview of each chapter. We provide installation, and execution in-
instructions for the examples in this book. We also discuss design patterns and how we use them to architect our examples.

**Chapter 2—Advanced Swing Graphical User Interface Components**

Advanced Swing components enable developers to build functionally rich user interfaces. The Swing graphical user interface components were introduced with the Java Foundation Classes (JFC) as a downloadable extension to Java 1.1 and became standard in the Java 2 Platform. Swing provides a much richer set of GUI components than Java’s original Abstract Windowing Toolkit (AWT), including advanced features such as a pluggable look-and-feel, lightweight component rendering and an enriched component set. This chapter introduces Swing components with which you can enrich your applicationGUIs.

Many of the examples in this chapter use the `JEditorPane` class extensively, which is capable of rendering styled content, such as HTML pages. We also present the first of our inline design patterns discussions. Swing `Actions` implement the Command design pattern to build reusable user interface logic. We also introduce useful Swing components such as `JSplitPane`, `JTabbedPane` and multiple-document-interface components for organizing GUI elements. Swing provides mechanisms for building applications for multiple languages and countries, and for disabled users. Building internationalized applications ensures that applications will be ready for use around the world in many languages. Accessibility ensures that users with disabilities will be able to use applications through commonly available utilities, such as screen readers. We show how developers can use Swing to build Java applications that are accessible to users in other countries and users with disabilities.

**Chapter 3—Model-View-Controller**

Advanced Swing components, including the `JTree` and `JTable` components enable developers to build flexible, data-driven graphical user interfaces in Java. The **Model-View-Controller (MVC)** architecture abstracts the GUI (the *view*) from the underlying data (the *model*). A *controller* determines how the application handles user interactions, such as mouse and keyboard events. The Swing components implement a variation of the MVC architecture that combines the view and controller to form a *delegate*. For example, a `JTree` is a delegate (i.e., combined view and controller) for its `TreeModel` (the model). The `TreeModel` contains the raw data to be presented in, and modified by, the `JTree`. The `JTree` provides a visual representation of the data and processes user interactions, such as renaming nodes in the tree. The benefit of this architecture is that each component can change without requiring changes in the other components. Furthermore, several delegates, views and controllers may be associated with a single model. MVC has many uses in desktop applications, enterprise applications, simulations and other types of programs. In this chapter, we discuss MVC in general and its variant, the *delegate-model architecture*. We also introduce the Observer design pattern, which is one part of the MVC architecture. After reading this chapter, you will be able to take advantage of advanced Swing components that use the delegate-model architecture, such as `JList`, `JTable`, and `JTree`.

**Chapter 4—Graphics Programming with Java 2D™ and Java 3D™**

The graphical features provided by the Java 2D API and the graphical user interface enhancements available in the Swing GUI components provide many tools for developing rich graphical content by incorporating line art, text and imaging in a single graphics model.
Developers can use these tools to build custom graphics and images as well as visual representations of data. The Java 2D API also provides advanced capabilities for text layout and manipulation. Imaging technology in the Java 2D API allows for manipulation of fixed resolution images, and includes filters for blurring and sharpening images as well as other image-processing tools. The Java 2D API also provides support for delivering graphical content to different devices by defining a logical coordinate system that is translated appropriately for a given output device such as a printer or monitor. We also introduce the Java 3D API for developing three-dimensional, virtual worlds in Java. The Java 3D API provides technologies for manipulating 3D objects. For example, the programmer can rotate, scale and translate 3D objects. Other advanced features include applying textures to 3D objects using texture mapping and varying the lighting effects on 3D objects by changing the colors and positions of light sources. We implement an application that allows the user to manipulate a 3D object with the mouse. We then present a substantial 3D game in which the user navigates a shape through a 3D scene full of “flying” obstacles. The goal of the game is to move this shape to a specific target point without colliding with any of the moving obstacles.

Chapter 5—Case Study: Java 2D Drawing Application with Design Patterns
The case study in this chapter implements a substantial Java application that integrates the many Java features and techniques presented in Chapters 2–4. We present a graphics application case study that uses the GUI capabilities of Chapters 2 and 3 and the two-dimensional graphics capabilities of Chapter 4, as well as the flexible capabilities of XML. The case study emphasizes the Model-View-Controller architecture (Chapter 3) to provide multiple views of a single drawing such as a detail view and a complete view. A multiple document interface (Chapter 2) allows users to modify multiple drawings in parallel. Swing Actions (Chapter 2) provide reusable user-interaction logic for menu and toolbar functionality. The case study also uses the Drag-and-Drop API to enable users to move shapes between drawings and to drop JPEG images onto a drawing from the local file system. We use several design patterns including the Factory Method, Adapter State and Template Method design patterns.

Chapter 6—JavaBeans Component Model
In this chapter, we take a deeper look into developing Java components based on the JavaBeans component architecture. JavaBeans (beans) allow developers to reap the benefits of rapid application development in Java by assembling predefined software components to create powerful applications and applets. Graphical programming and design environments (often called builder tools) that support beans provide programmers with tremendous flexibility by allowing programmers to reuse existing components. A programmer can integrate these components to create applets, applications or even new beans for reuse by others. JavaBeans and other component-based technologies have led to a new type of programmer—the component assembler, who uses pre-built components to create richer functionality. Component assemblers do not need to know the implementation details of components, but they need to know what services the components provide. Component assemblers can make beans communicate through the beans’ well-defined services (i.e., methods), typically without writing any code (the builder tool often generates code, which is sometimes hidden from the component assembler—depending on the tool). Indeed, a component assembler can create complex applications simply by “connecting the dots.”
This chapter shows you how to use existing beans and how to create new beans. After studying this chapter, you will have a foundation in JavaBeans programming that will enable you to develop applications and applets rapidly using the more advanced features of integrated development environments that support beans.

Chapter 7—Security
Security is a primary concern in the development of software systems. This chapter discusses the issues associated with security and introduces Java technologies that ensure successful, secure transactions. Among these technologies is the Java Cryptography Extension (JCE), which supports secret-key encryption and digital signatures. The Java Secure Socket Extension (JSSE) supports the Secure Sockets Layer (SSL) protocol—one of the most widely used tools for securing Internet communications. JSSE provides encryption, message integrity checks and authentication of servers and clients. Java also provides the Java Authentication and Authorization Service (JAAS) for authenticating users and granting permissions. The basis for Java security is the sandbox security model in which applets and applications execute. The sandbox is a protected environment that prevents Java programs from accessing protected resources. The program must be granted specific permissions to access system resources, such as the files on a user’s computer and servers on the Internet. Permissions may be granted through policy files.

Chapter 8—Java Database Connectivity (JDBC)
Access and storage of data are integral to creating powerful software applications. This chapter discusses Java’s support of database manipulation. Today’s most popular database systems are relational databases. We present examples using Cloudscape 3.6.4—a pure-Java database management system from Informix Software. Cloudscape is available free for download (for learning and development purposes) at www.cloudscape.com and is on the CD that accompanies this book. Java programmers communicate with databases and manipulate their data using the Java Database Connectivity (JDBC) API. A JDBC driver implements the interface to a particular database. This chapter introduces JDBC and uses it to connect to a Cloudscape database, then to manipulate its content. We use the Structured Query Language (SQL) to extract information from, and insert information into, a database. We then use JDBC and SQL to create an address-book application that stores, updates and deletes addresses. Several later chapters use the techniques shown in this chapter to build data-driven Web and enterprise applications.

Chapter 9—Servlets
Servlets extend the functionality of servers—typically Web servers. Servlets are effective for developing Web-based solutions that interact with databases on behalf of clients, dynamically generate custom content to be displayed by browsers, and maintain unique session information for each client. Many developers feel that servlets are the right solution for database-intensive applications that communicate with so-called thin clients—applications that require minimal client-side processing capability. Clients connect to the server using standard protocols, such as HyperText Transfer Protocol (HTTP), available on most client platforms through Web browsers (and other applications). Thus, the application logic can be written once and reside on the server for access by clients. The Java Servlet API allows developers to add functionality to Web servers for handling client requests. Unlike the Common Gateway Interface (CGI), in which a separate process may be started
for each client request, servlets typically are threads in a single JVM process. Servlets also are reusable across Web servers and across platforms. This chapter demonstrates the Web’s request/response mechanism (primarily with HTTP get and post requests), session-tracking capabilities, redirecting requests to other resources and interacting with databases through JDBC.

Chapter 10—Java Server Pages (JSP)
This chapter introduces an extension of servlet technology called Java Server Pages (JSP). JSPs enable delivery of dynamically generated Web content and are used primarily for developing presentation logic in Enterprise Java applications. JSPs may contain Java code in the form of scriptlets and may also use JavaBeans components. Custom tag libraries enable Web-page designers unfamiliar with Java to enhance Web pages with powerful dynamic content and processing capabilities created by Java developers. To increase performance, each JSP is compiled into a Java Servlet—this normally occurs the first time each JSP is requested by a client. Subsequent client requests are fulfilled by the compiled servlet.

Chapter 11—Case Study: Servlets and JSP Bookstore
This chapter is a capstone for our presentation of JSPs and servlets. Here, we implement a bookstore Web application that integrates JDBC, XML, JSP and servlet technologies. We discuss additional servlet features as they are encountered in the case study. This chapter deploys the bookstore application on the J2EE 1.2.1 reference implementation application server software. The J2EE 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container. After reading this chapter, you will be able to implement a substantial distributed Web application with many components, and you will be able to deploy that application on the J2EE 1.2.1 application server.

Chapter 12—Java-Based Wireless Applications Development and J2ME
One topic of particular interest in e-business and e-commerce applications is wireless Internet technology. Wireless technology turns e-business into m-business, or mobile business. It allows you to connect to the Internet any time from almost any place. You can use it to conduct online transactions, make purchases, trade stocks and send e-mail. New technologies already enable the wireless office, where computers, phones and other office equipment are networked without cables. This chapter introduces some of the more popular wireless technologies, including WAP, i-mode and the Java 2 Platform Micro Edition™ (J2ME). J2ME brings Java technology to embedded devices and consumer devices that have limited processing power and memory. J2ME includes specialized APIs for many consumer devices, including cellular phones, smart cards, Internet appliances and PDAs (personal digital assistants), such as Palm™ and PocketPC. The K Virtual Machine—a trimmed-down version of the Java virtual machine for consumer devices—provides the essential features for executing Java code on these devices. Using servlets and XML, we present a case study of a three-tier application that sends a game for several wireless device types.

Chapter 13—Remote Method Invocation (RMI)
This chapter introduces Remote Method Invocation (RMI)—a technology for building distributed systems in Java. Using RMI, Java objects can be located on computers across a network, yet still interact as if they resided on a single computer. Java objects can perform lookups to find remote objects on the network and invoke methods across a local area net-
work (LAN) or even the Internet. RMI allows Java-object-to-Java-object distributed communication. Once a Java object registers as being remotely accessible (i.e., it is a remote object), a client can “look up” that Java object and obtain a reference that allows the client to use that object remotely. The method call syntax is identical to the syntax for calling methods of other objects in the same program. RMI handles the marshalling (i.e., collecting and packaging) of data across the network; RMI also enables Java programs to transfer complete Java objects using Java’s object-serialization mechanism. The programmer need not be concerned with the details of transmitting data over the network.

Chapter 14—Session Enterprise JavaBeans (EJBs) and Distributed Transactions

Enterprise JavaBeans (EJBs) enable Java developers to build robust multi-tier applications. In a multi-tier application the responsibilities of providing services to a client can be divided among multiple servers. A typical two-tier application consists of the client-tier and the server-tier. A three-tier architecture often makes use of an application server as a middle-tier between the client Web browser and a database server. Enterprise JavaBeans provide a framework for building middle-tier business-logic implementations. Using RMI and EJB Containers, Enterprise JavaBeans also allow for business logic to be distributed across a network. We introduce Enterprise JavaBeans (EJBs), which provide a component model for building business logic in enterprise Java applications. We discuss session EJBs in their two forms: stateful and stateless. We demonstrate how to develop both stateless and stateful session EJBs. We also introduce EJB support for distributed transactions, which help to ensure data integrity across databases and across application servers. We show how to build EJBs that take advantage of J2EE’s distributed transaction support to update data across multiple databases atomically.

Chapter 15—Entity Enterprise JavaBeans (EJBs)

This chapter continues our discussion of Enterprise JavaBeans with an introduction to entity Enterprise JavaBeans. Unlike session EJBs, entity EJBs store data in long-term storage, such as in a database. Entity EJBs provide an object-oriented representation of persistent data, such as data stored in an RDBMS or legacy application. Entity EJBs can be used to build powerful and flexible data applications. There are two types of entity EJBs—those that use bean-managed persistence and those that use container-managed persistence. Entity EJBs that use bean-managed persistence implement code for storing and retrieving data from the persistent data sources they represent. For example, an entity EJB that uses bean-managed persistence might use the JDBC API to store and retrieve data in a relational database. Entity EJBs that use container-managed persistence rely on the EJB container to implement the data-access calls to their persistent data sources. The developer must supply information about the persistent data source when deploying the EJB. This chapter provides a demonstration of both types of entity EJBs.

Chapter 16—Java Message Service (JMS)

The Java Message Service (JMS) provides an API for integrating enterprise Java applications with message-oriented middleware (MOM) systems. Message-oriented middleware enables applications to communicate by sending messages to one another. Message-oriented middleware is a popular technology for building loosely coupled applications. This chapter introduces the two basic messaging system models—point-to-point and publish/subscribe. We demonstrate Java’s interfaces for both of these models. We also provide an introduction to message-driven EJBs—a new feature of J2EE version 1.3.
Chapter 17—E-Business Case Study: Architectural Overview
The technologies that comprise the Java 2 Enterprise Edition (J2EE) enable developers to build robust, scalable enterprise applications. In this case study, we build an e-business application using several features of J2EE, including servlets, Enterprise JavaBeans, XML and XSLT. We also integrate wireless technology, including WAP/WML and i-mode/cHTML. In this chapter, we present an overview of the Deitel Bookstore case study architecture, which uses the MVC design pattern in an enterprise application context. In the following chapters, we present the controller logic implementation with servlets (Chapter 18) and the business logic and data abstraction implementation with EJBs (Chapters 19 and 20).

Chapter 18—E-Business Case Study: Presentation and Controller Logic
This chapter presents the implementation of the controller and presentation logic for the Deitel Bookstore case study. Controller logic in an application is responsible for handling user requests. The Java servlets in the Deitel Bookstore implement the controller logic for the application. Every user request is handled by a servlet that takes the appropriate action, based on the request type (e.g., a request to view the store catalog) and presents content to the client. We use XSLT transformations to implement the presentation logic for the application—the view in MVC. After invoking business-logic methods to process a client request, the servlets generate XML documents that contain content to be presented to the client. These XML documents are not specific to any particular type of client (e.g., Web browser, cell phone, etc.); they simply describe the data supplied by the business logic. An XSL transformation is applied to the XML documents to present the information to the user in the appropriate format. For example, an XSL transformation might generate an XHTML document to present to a Web browser, or a WML document to present to a WAP browser. XSL transformations are needed for each type of client the application supports. We could enable the application to support other types of clients simply by implementing additional sets of style sheets and editing a configuration file.

Chapter 19—E-Business Case Study: Business Logic Part I
In this chapter, we present the EJB business logic for the shopping-cart e-business model and entity EJBs for maintaining product inventory of the Deitel Bookstore case study. The primary goal of an on-line store application is to enable customers to purchase products. EJB business logic implements the business rules that govern this process. We implement the business logic for managing the set of products a customer wishes to purchase as a ShoppingCart EJB. The ShoppingCart EJB enforces business rules that define how products are added to the shopping cart, how shopping carts are created and how customers complete their purchases. We also present entity EJBs that represent on-line store products and orders. After reading this chapter, you will understand the use of EJBs in an e-business application context, as well as more advanced EJB topics, such as custom primary-key classes and many-to-many relationships.

Chapter 20—Enterprise Java Case Study: Business Logic Part 2
In this chapter we present the business logic for managing customers in our Deitel Bookstore case study. Maintaining information about the customers of an online store can make purchases more convenient by storing billing and shipping information on the server. The online store’s marketing department may also use gathered data for distribution of marketing materials and analyzing demographic information. We also present an entity EJB that
generates unique IDs for the Customer, Order and Address EJBs. Instances of these EJBs are created when new customer’s register and when customer’s place new orders. Relational databases require unique primary keys to maintain referential integrity and perform queries. We provide the `SequenceFactory` EJB to generate these unique IDs because not all databases can generate these primary-key values automatically.

**Chapter 21—Application Servers**
This chapter introduces several commercial application servers—an application server is software that integrates server-side logic components to allow communication between components and tiers of a software architecture. Application servers also manage the persistence, life cycles, security and various other services for logic components. We discuss the concepts behind application servers and introduce three popular commercial application servers, including BEA’s WebLogic, IBM’s WebSphere and the iPlanet Application Server. We present a detailed walkthrough of deploying the Deitel Bookstore application on BEA’s WebLogic and IBM’s WebSphere, both of which we include on the CD-ROM that accompanies this book. As we went to publication, iPlanet was about to release a new version of their application server. Please visit [www.iplanet.com/ias_deitel](http://www.iplanet.com/ias_deitel) to download the latest version. We also will provide complete deployment instructions for the Deitel Bookstore case study on iPlanet at our Web site, [www.deitel.com](http://www.deitel.com).

**Chapter 22—Jini**
Jini Technology is an advanced set of network protocols, programming models and services that enable true plug-and-play interactions between networked Jini-enabled devices and software components. Jini technology allows distributed-systems developers to discover and use Jini-enabled resources on the network. The heart and soul of Jini comes from its robust and standardized network protocols, including multicast request protocol, multicast announcement protocol and unicast discovery protocol. Jini-enabled resources—or services—use these three protocols to locate and interact with other services. Beyond the network protocols, Jini technology provides the infrastructure required to use the protocols. This infrastructure exists as a set of classes that hide the low-level details of the protocols, allowing developers to focus on functionality instead of implementation. This chapter overviews Jini technology, introduces the network protocols that support Jini services and demonstrates Jini technology with a substantial Jini application. Later in the book (Chapter 29, Peer-to-Peer Applications and JXTA) we use Jini to build and instant-messaging application.

**Chapter 23—JavaSpaces**
Objects that take part in distributed systems must be able to communicate with one another and share information. The `JavaSpaces service` is a Jini service that implements a simple, high-level architecture for building distributed systems using a distributed repository for objects and three simple operations—`read`, `write` and `take`. JavaSpaces services support transactions through the Jini transaction manager, and a notification mechanism that notifies an object when an entry that matches a given template is written to the JavaSpaces service. In the first half of this chapter, we present fundamental JavaSpaces technology concepts and use simple examples to demonstrate operations, transactions and notifications. The case study at the end of this chapter uses JavaSpaces services to build an image-processing application that distributes the work of applying filters to images across many programs on separate computers.
Chapter 24—Java Management Extensions (JMX) (on CD)
This chapter introduces the Java Management Extensions (JMX), which were developed by Sun and other network-management industry leaders to define a component framework for building intelligent network-management applications. JMX defines a three-level management architecture—**instrumentation level, agent level** and **manager level**. The instrumentation level allows clients to interact with objects (called **managed resources**) by exposing public interfaces to those objects. The agent level contains **JMX agents**, which enable communication between remote clients and managed resources. The manager level contains applications (clients) that access and interact with managed resources via the JMX agents. JMX also provides support for existing management protocols—such as SNMP—so developers can integrate JMX solutions with existing management applications. This chapter discusses JMX architecture and presents a case study that uses JMX capabilities to manage a network printer simulator.

Chapter 25—Jiro (on CD)
This chapter serves as an introduction to Sun’s Jiro technology, a Java-based technology that provides infrastructure for developing management solutions for distributed resources on heterogeneous networks. Jiro is an implementation of the Federated Management Architecture (FMA) specification, which defines a standard protocol for communication between heterogeneous managed resources (such as devices, systems, applications). Jiro technology supports a three-tier architecture of management solutions. The top tier is the **client tier**. The client locates and communicates with the management services. The **middle tier** provides both static and dynamic management services. The **bottom tier** consists of the heterogeneous managed resources. Jiro is a complementary technology to JMX and can be used to build management solutions. The chapter concludes with a similar case study to the JMX case study presented in Chapter 24.

Chapter 26—Common Object Request Broker Architecture (CORBA): Part 1 (on CD)
In this chapter, we introduce the Common Object Request Broker Architecture (CORBA). CORBA is an industry-standard, high-level distributed object framework for building powerful and flexible service-oriented applications. We investigate the essential details of CORBA as defined in the Object Management Group (OMG) specifications. We discuss the **Object Request Broker (ORB)**—the core of the CORBA infrastructure—and describe how it makes CORBA a powerful distributed object framework. We discuss the **Java Interface Definition Language (JavaIDL)**—the official mapping of Java to CORBA. Live-code examples demonstrate how to write CORBA-compliant distributed code using Java. Both client-side and server-side JavaIDL are demonstrated. A feature of the chapter is a case study that implements the Deitel Messenger application using CORBA.

Chapter 27—Common Object Request Broker Architecture (CORBA): Part 2 (on CD)
This chapter continues the discussion of CORBA. We introduce the Dynamic Invocation Interface as well as CORBA services, including the Naming, Security, Object Transaction and Persistent State services. The discussion continues with a comparison of RMI and CORBA; we also introduce RMI-IIOP, used to integrate RMI with CORBA. Finally, we present an alternate implementation of the Deitel Messenger application using RMI-IIOP.
Chapter 28—Peer-to-Peer Applications and JXTA
Instant-messaging applications and document-sharing systems such as AOL Instant Messenger™ and Gnutella have exploded in popularity, transforming the way users interact with one another over networks. In a peer-to-peer (P2P) application, each node performs both client and server functions. Such applications distribute processing responsibilities and information to many computers, thus reclaiming otherwise wasted computing power and storage space, and eliminating central points of failure. In this chapter, we introduce the fundamental concepts of peer-to-peer applications. Using Jini (Chapter 22), RMI (Chapter 13) and multicast sockets, we present two peer-to-peer application case studies of instant-messaging systems. The first implementation uses Jini and RMI, and the second uses multicast sockets and RMI. Finally, we introduce JXTA (short for “juxtapose”)—a new open-source technology from Sun Microsystems™ that defines common protocols for implementing peer-to-peer applications.

Chapter 29—Introduction to Web Services with SOAP
Interoperability, or seamless communication and interaction between different software systems, is a primary goal of many businesses and organizations that rely heavily on computers and electronic networks. This chapter introduces Web services with Simple Object Access Protocol (SOAP), a protocol designed to address this issue. Web services can be Web accessible applications, such as Web pages with dynamic content. More specifically, Web services expose public interfaces for Web-based applications to use. SOAP is a protocol that uses XML to make remote-procedure calls over HTTP to provide interoperability between disparate Web-based applications.

Appendix A—Creating Markup with XML (on CD)
XML is enormously important in Advanced Java 2 Platform How to Program and is integrated into examples throughout the book. We have included a substantial introduction to XML in Appendices A–D. Appendix A introduces the fundamentals of XML. We discuss the properties of the XML character set, called Unicode—the standard aimed at providing a flexible character set for all the world’s languages. (Appendix I introduces Unicode.) We provide a brief overview of parsers—programs that process XML documents and their data. We also overview the requirements for a well-formed document (i.e., a document that is syntactically correct). We discuss elements, which hold data in XML documents. Several elements can have the same name (resulting in naming collisions); we introduce namespaces, which differentiate these elements to avoid these collisions.

Appendix B—XML Document Type Definitions (on CD)
A Document Type Definition (DTD) is a structural definition for an XML document, specifying the type, order, number and attributes of the elements in an XML document as well as other information. By defining the structure of an XML document, a DTD reduces the validation and error-checking work of the application using the document. We discuss well-formed and valid documents (i.e., documents that conform to a DTD). This appendix shows how to specify different element and attribute types, values and defaults that describe the structure of the XML document.
Appendix C—XML Document Object Model (DOM) (on CD)
The W3C Document Object Model (DOM) is an API for XML that is platform and language independent. The DOM API provides a standard API (i.e., methods, objects, etc.) for manipulating XML-document contents. The Java API for XML Processing (JAXP) provides DOM support for Java programs. XML documents are hierarchically structured, so the DOM represents XML documents as tree structures. Using DOM, programs can modify the content, structure and formatting of documents dynamically. This appendix examines several important DOM capabilities, including the ability to retrieve data, insert data and replace data. We also demonstrate how to create and traverse documents using the DOM.

Appendix D—XSLT: Extensible Stylesheet Language Transformations (on CD)
XSL was designed to manipulate the rich and sophisticated data contained in an XML document. XSL has two major functions: formatting XML documents and transforming them into other data formats such as XHTML, Rich Text Format (RTF), etc. In this appendix, we discuss the subset of XSL called XSLT. XSLT uses XPath—a language of expressions for accessing portions of XML documents—to match nodes for transforming an XML document into another text document. We use JAXP—which includes XSLT support—in our examples. An XSL stylesheet contains templates with which elements and attributes can be matched. New elements and attributes can be created to facilitate a transformation.

Appendix E—Downloading and Installing J2EE (on CD)
We use the Java 2 Enterprise Edition extensively in this book to create substantial enterprise applications. This appendix provides instructions for downloading and installing Sun’s reference implementation of the J2EE.

Appendix F—Java Community Process (JCP) (on CD)
This appendix provides an overview of the Java Community Processes (JCP), which Sun Microsystems started in 1998. The JCP (www.jcp.org) allows Java individuals, organizations and corporations to participate in the development of new technologies and APIs for the Java Platform. Sun has integrated a number of technologies developed through the Java Community Process into the Java 2 Platform Software Development Kits, including the XML parsing specification.

Appendix G—Java Native Interface (JNI) (on CD)
The Java Native Interface (JNI) allows programmers to access pre-built applications and libraries written in other languages. JNI allows programmers to work in Java without requiring developers to rebuild existing libraries. JNI can be useful in time-critical applications—programmers may write a piece of the application in assembly code and link this program with Java to provide better performance. In this appendix, we explain how to integrate Java with C++ libraries. Included are the most common uses and functions of JNI. We show how Java programs can call native functions stored in compiled libraries, and how native code can access Java objects, methods and member variables from C++. Understanding these examples requires familiarity with C++.

Appendix H—Career Opportunities (on CD)
The Internet presents valuable resources and services for job seekers and employers. Automatic search features allow employees to scan the Web for open positions. Employers also
can find job candidates using the Internet. This greatly reduces the amount of time spent preparing and reviewing resumes, as well as travel expenses for distance recruiting and interviewing. In this chapter, we explore career services on the Web from the perspectives of job seekers and employers. We introduce comprehensive job sites, industry-specific sites (including site geared specifically for Java and wireless programmers) and contracting opportunities.

Appendix I—Unicode (on CD)
This appendix introduces the Unicode Standard—a character-set-encoding standard that facilitates the production and distribution of software. As computer systems evolved worldwide, computer vendors developed numeric representations of character sets and special symbols for the local languages in different countries. In some cases, different representations were developed for the same languages. Such disparate character sets made communication between computer systems difficult. XML and XML-derived languages, such as WML, support the Unicode Standard, which defines a single character set with unique numeric values for characters and special symbols for most of the world’s languages. In this appendix, we discuss the Unicode Standard and the Unicode Consortium (www.unicode.org)—a non-profit organization that maintains the Unicode Standard.

1.4 Running Example Code
Many example programs in Advanced Java 2 Platform How to Program are quite complex and require special software to execute. For example, Chapters 17–20 present a J2EE case study that requires an application server, which provides a runtime environment and services for an enterprise application. This case study also requires a database. For this and many other programs we provide installation, deployment and execution instructions in the text and at our Web site, www.deitel.com.

At the time of this writing, Java 2 Enterprise Edition reference implementation version 1.2.1 was the current, released version of J2EE, and version 1.3 was in beta release. We will update installation instructions on our Web site when Sun releases version 1.3, which will include several enhancements and updates. For example, version 1.3 implements the Java messaging Service (JMS 1.0.2), J2EE Connector Technology and the Java API for XML Processing (JAXP 1.1). Java Servlets (version 2.3) implement filters, a lightweight transfer framework for requests and responses, monitoring application lifecycles and better internationalization support. The Java Server Pages implementation (version 1.2) features improved runtime support for tag libraries and translation time JSP page validation. The 1.3 Enterprise JavaBeans implementation (EJB 2.0) supports message-driven enterprise beans, interoperability between EJB containers and Container-Managed Persistence 2.0.

The examples in Advanced Java 2 Platform How to Program use Sun’s standard naming convention for packages. We place each example in an appropriately named sub-package of package com.deitel. For example, the WebBrowser example in Chapter 2, Advanced Swing Graphical User Interfaces, contains the package declaration

    package com.deitel.advjhtp1.gui.webbrowser;

The acronym advjhtp1 in the package name indicates that this package is from Advanced Java 2 Platform How to Program, First Edition. This package structure requires that you compile the examples into the corresponding directory structure.
Managing packages with Java’s command-line compiler and tools can be cumbersome, so we recommend that readers use an integrated development environment to simplify the development and execution of the examples and exercises in this book. We used Sun’s Forte for Java Community Edition—which derives from the open-source NetBeans IDE (www.netbeans.org)—to develop the code examples for this book. We have included Forte for Java, Community Edition version 2.0 and the Java 2 Standard Edition SDK version 1.3.1 on the CD that accompanies this book. For tutorials on how to install Forte and how to develop applications with it, please refer to Forte’s help system or the documentation at:

www.sun.com/forte/ffj/documentation/index.html

Most Java development environments enable developers to load directory structures containing Java packages directly into those environments. To facilitate working with the code in this way, we have provided the complete directory structure, with source files in the appropriate locations, on the CD-ROM that accompanies this book. We recommend that you copy this directory structure from the CD-ROM that accompanies this book to your hard drive. Once you have copied the directory structure, you can load the examples according to the instructions for your development environment.

For readers who wish to use command-line tools for compiling and executing the programs in this book, we also provide separate folders with the examples for each chapter. To compile and execute the examples from the command line, copy the folder for the particular chapter or example onto your hard drive. For example, if you copy the ch02 directory to the C:\examples directory on your hard drive, you can compile the WebBrowser example using the commands

```
cd C:\examples\ch02\fig02_01
javac -d . WebBrowser.java WebBrowserPane.java WebToolBar.java
```

The command-line argument `-d .` specifies that the Java compiler should create the resulting .class files in the appropriate directory structure. To execute the example, you must provide the fully qualified package name for the class that defines method `main`. For example,

```
java com.deitel.advjhtp1.gui.webbrowser.WebBrowser
```

### 1.5 Design Patterns

Most code examples presented in introductory Java books—such as our `Java How to Program, Fourth edition`—contain fewer than 150 lines of code. These examples do not require an extensive design process, because they use only a few classes and illustrate rudimentary programming concepts. However, most of the programs in `Advanced Java How to Program`, such as the Java 2D case study (Chapter 5), the three-tier Wireless application (Chapter 12) and the Deitel Bookstore (Chapters 17–20), are much more complex. Such large applications can require thousands of lines of code, contain many interactions among objects and involve many user interactions. For such software, it is important to employ proven, effective design strategies. Systems such as automated-teller machines and air-traffic control systems can contain millions, or even hundreds of millions, of lines of code. Effective design is absolutely crucial to the proper construction of such complex systems.
Over the past decade, the software engineering industry has made significant progress in the field of design patterns—proven architectures for constructing flexible and maintainable object-oriented software.\footnote{Using design patterns can reduce the complexity of the design process substantially. Well-designed object-oriented software allows designers to reuse and integrate pre-existing components into future systems. Design patterns benefit system developers by

- helping to construct reliable software using proven architectures and accumulated industry expertise
- promoting design and code reuse in future systems
- helping to identify common mistakes and pitfalls that occur when building systems
- helping to design systems independently of the languages in which they will ultimately be implemented
- establishing a common design vocabulary among developers
- shortening the design phase in a software-development process}

The notion of using design patterns to construct software systems originated in the field of architecture. Architects use established architectural design elements, such as arches and columns, when designing buildings. Designing with arches and columns is a proven strategy for constructing sound buildings—these elements may be viewed as architectural design patterns.

\subsection*{1.5.1 History of Object-Oriented Design Patterns}

During 1991–1994, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides—collectively known as the “gang of four”—combined their expertise in writing the book Design Patterns, Elements of Reusable Object-Oriented Software (Addison-Wesley: 1995). This book showed that design patterns evolved naturally through years of industry experience. John Vlissides states that “the single most important activity in pattern writing is reflection.”\footnote{This statement implies that to create patterns, developers must reflect on, and document, their successes (and failures) when designing and implementing software systems. Developers use design patterns to capture and use this collective experience, which ultimately helps them share similar successes with other developers.}

The gang-of-four book described 23 design patterns, each providing a solution to a common software design problem. The book groups design patterns into three categories—creational, structural and behavioral design patterns. Figure 1.1 lists these design patterns.

Creational design patterns describe techniques for instantiating objects (or groups of objects). These design patterns address issues related to the creation of objects, such as preventing a system from creating more than one object of a class (e.g., Singleton) or deferring until execution time the decision as to what types of objects are created (e.g., Factory Method). For example, suppose we are designing a 3-D drawing program, in which the user can create several 3-D geometric objects, such as cylinders, spheres, cubes, tetrahedrons, etc. At compile time, the program does not know what types of shapes the user will choose to add to the drawing. Based on user input at runtime, this program should determine the class from which to instantiate an object. If the user chooses to create a cylinder, the program should “know” to instantiate an object of class \texttt{Cylinder} and add it to the drawing.
When the user decides what geometric object to draw, the program should determine the specific subclass from which to instantiate that object.

Structural design patterns describe common ways to organize classes and objects in a system. Developers often find two problems with poor organization. The first is that classes are assigned too many responsibilities. Such classes may damage information hiding and violate encapsulation, because each class may have access to information that belongs in a separate class. The second problem is that classes can overlap responsibilities. Burdening a design with unnecessary classes wastes time for designers because they will spend hours trying to extend or modify classes that should not even exist in the system. As we will see, structural design patterns help developers avoid these problems.

Behavioral design patterns assign responsibilities to objects. These patterns also provide proven strategies to model how objects collaborate with one another and offer special behaviors appropriate for a wide variety of applications. The Observer pattern is a classic example of collaborations between objects and of assigning responsibilities to objects. For example, GUI components use this patterns to communicate with their listeners, which respond to user interactions. A listener observes state changes in a particular component by registering to handle that component’s events. When the user interacts with the component, that component notifies its listeners (also known as its observers) that the component’s state has changed (e.g., a button has been pressed).

### 1.5.2 Design Patterns Discussion

Design patterns are implemented in code as sets of classes and objects. To use design patterns effectively, designers must familiarize themselves with the most popular and effective patterns used in the software-engineering industry. Throughout this book, we discuss fundamental object-oriented design patterns and architectures, as well as their importance in constructing well-engineered software. We discuss each design pattern as it is used in a particular code example or case study. Figure 1.2 lists those design patterns that we used and in which chapter we used them.

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Chain-of-Responsibility</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td>Factory Method</td>
<td>Composite</td>
<td>Iterator</td>
</tr>
<tr>
<td>Prototype</td>
<td>Decorator</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Singleton</td>
<td>Facade</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Template Method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>

**Fig. 1.1** Gang-of-four 23 design patterns.
Note that Fig. 1.2 does not include every design pattern specified in Fig. 1.1. We used only those patterns that were appropriate for solving specific design problems that we encountered when writing the examples and case studies in this book. We now list other popular “gang-of-four” design patterns that are useful in building software, even though we did not use them when building the examples for this book.

**Prototype**

Sometimes, a system must make a copy of an object but will not know that object’s class until run time. For example, consider a drawing program that contains several “shape” classes (e.g., classes `Line`, `Oval` and `Rectangle`, etc.) that extend an abstract superclass `Shape`. The user of this program should, at any time, be able to create, copy and paste new instances of `Shape` classes to add those shapes to drawings. The **Prototype design pattern** enables the user to accomplish this. This design pattern allows an object—called a **prototype**—to clone itself. The prototype is similar to a rubber stamp that can be used to create several identical “imprints.” In software, every prototype must belong to a class that implements a common interface that allows the prototype to clone itself. For example, the Java API provides method `clone` from interface `java.lang.Cloneable`—any object from a class that implements interface `Cloneable` uses method `clone` to make a copy of itself. Specifically, method `clone` creates a copy of an object, then returns a reference to that object. In the drawing program, if we designate class `Line` as the prototype, then it should implement interface `Cloneable`. To create a new line in our drawing, we clone the `Line` prototype—this prototype will return a reference to a different `Line` object. To copy a preexisting line, we clone that `Line` object. Developers often use method `clone` to prevent altering an object through its reference, because method `clone` returns a reference to an object’s copy, rather than return the object’s actual reference.

**Bridge**

Suppose we are designing a `Button` class for both the Windows and Macintosh operating systems. Class `Button` contains specific button information such as an `ActionListener` and a `String` label. We design classes `Win32Button` and `MacButton` to extend class `Button`. Class `Win32Button` contains “look-and-feel” information on how to display a
Button on the Windows operating system, and class MacButton contains “look-and-feel” information on how to display a Button on the Macintosh operating system.

Two problems arise from this approach. First, if we create new Button subclasses, we must create corresponding Win32Button and MacButton subclasses. For example, if we create class ImageButton (a Button with an overlapping Image) that extends class Button, we must create additional subclasses Win32ImageButton and MacImageButton. In fact, we must create Button subclasses for every operating system we wish to support, which increases development time. The second problem is that when a new operating system enters the market, we must create additional Button subclasses specific to that operating system.

The Bridge design pattern avoids these problems by separating an abstraction (e.g., a Button) and its implementations (e.g., Win32Button, MacButton, etc.) into separate class hierarchies. For example, the Java AWT classes use the Bridge design pattern to enable designers to create AWT Button subclasses without needing to create corresponding subclasses specific to each operating system. Each AWT Button maintains a reference to a ButtonPeer, which is the superclass for platform-specific implementations, such as Win32ButtonPeer, MacButtonPeer, etc. When a programmer creates a Button object, class Button determines which platform-specific ButtonPeer object to create and stores a reference to that ButtonPeer object—this reference is the “bridge” in the Bridge design pattern. When the programmer invokes methods on the Button object, the Button object invokes the appropriate method on its ButtonPeer object to fulfill the request. If a designer creates Button subclass ImageButton, the designer does not need to create a corresponding Win32ImageButton or MacImageButton class. Class ImageButton “is a” Button, so when a programmer invokes an ImageButton method—such as setImage—on an ImageButton object, the Button superclass translates that method invocation into an appropriate ButtonPeer method invocation—such as drawImage.

Portability Tip 1.1

Designers often use the Bridge design pattern to enhance the platform independence of their systems. We can design Button subclasses without worrying about how an operating system implements each subclass.

Iterator

Designers use data structures such as arrays, linked lists and hash tables to organize data in a program. The Iterator design pattern allows objects to access individual objects from data structures without knowing that data structure’s implementation or how it stores object references. Instructions for traversing the data structure and accessing its elements are stored in a separate object called an iterator. Each data structure has an associated iterator implementation capable of traversing that data structure. Other objects can use this iterator, which implements a standard interface, regardless of the underlying data structure or implementation. Interface Iterator from package java.util uses the Iterator design pattern. Consider a system that contains Sets, Vectors and Lists. The algorithm for retrieving data from each structure differs among the classes. With the Iterator design pattern, each class contains a reference to an Iterator that stores traversal information specific to each data structure. For objects of these classes, we invoke an object’s iterator method to obtain a reference to an Iterator for that object. We invoke method next of the
**Iterator** to receive the next element in the structure without having to concern ourselves with the details of traversal implementation.

**Memento**
Consider a drawing program that allows a user to draw graphics. Occasionally the user may position a graphic improperly in the drawing area. The program can offer an “undo” feature that allows the user to unwind such errors. Specifically, the program would restore the drawing area’s original state (before the user placed the graphic). More sophisticated drawing programs offer a *history*, which stores several states in a list, so the user can restore the program to any state in the history. The *Memento design pattern* allows an object to save its state, so that—if necessary—the user can restore the object to its former state.

The Memento design pattern requires three types of objects. The *originator object* occupies some *state*—the set of attribute values at a specific time in program execution. In our drawing-program example, the drawing area is the originator, because it occupies several states. The drawing area’s initial state is that the area contains no elements. The *memento object* stores a copy of all attributes associated with the originator’s state. The memento is stored as the first item in the history list, which acts as the *caretaker object*—the object that contains references to all memento objects associated with the originator.

Now, suppose the user draws a circle in the drawing area. The area now occupies a different state—the area contains a circle object centered at specified *x-y* coordinates. The drawing area then uses another memento to store this information. This memento is stored as the second item in the history list. The history list displays all mementos on screen, so the user can select which state to restore. Suppose the user wishes to remove the circle—if the user selects the first memento from the list, the drawing area uses the first memento to restore itself to a blank area.

**Strategy**
Package *java.awt* offers several *LayoutManager* s, such classes *FlowLayout*, *BorderLayout* and *GridLayout*, with which developers build graphical user interfaces. Each *LayoutManager* arranges GUI components in a *Container* object—however, each *LayoutManager* implementation uses a different algorithm to arrange these components. A *FlowLayout* arranges components in a left-to-right sequence, a *BorderLayout* places components into five distinct regions and a *GridLayout* arranges components in row-column format. Interface *LayoutManager* plays the role of the *strategy* in the *Strategy design pattern*.

The Strategy design pattern allows developers to encapsulate a set of algorithms—called a strategy—that each have the same function (e.g., arrange GUI components) but different implementations. For example, interface *LayoutManager* (the strategy) is the set of algorithms that arranges GUI components. Each concrete *LayoutManager* subclass (e.g., the *FlowLayout*, *BorderLayout* and *GridLayout* objects) implements method *addLayoutComponent* to provide a specific component-arrangement algorithm.

### 1.5.3 Concurrency Patterns

Many additional design patterns have been created since the publication of the gang-of-four book, which introduced patterns involving object-oriented systems. Some of these new patterns involve specific types of object-oriented systems, such as concurrent, distributed or parallel systems. Multithreaded programming languages such as Java allow designers to
specify concurrent activities—that is, activities that operate in parallel with one another. Improper design of concurrent systems can introduce concurrency problems. For example, two objects attempting to alter shared data at the same time could corrupt that data. In addition, if two objects wait for one another to finish tasks, and if neither can complete their task, these objects could potentially wait forever—a situation called deadlock. Using Java, Doug Lea and Mark Grand created a set of concurrency patterns for multithreaded design architectures to prevent various problems associated with multithreading. We provide a partial list of these design patterns:

- The Single-Threaded Execution design pattern prevents several threads from invoking the same method of another object concurrently. In Java, developers can use the synchronized keyword to apply this pattern.

- The Balking design pattern ensures that a method will balk—that is, return without performing any actions—if an object occupies a state that cannot execute that method. A variation of this pattern is that the method throws an exception describing why that method is unable to execute—for example, a method throwing an exception when accessing a data structure that does not exist.

- The Read/Write Lock design pattern allows multiple threads to obtain concurrent read access on an object but prevents multiple threads from obtaining concurrent write access on that object. Only one thread at a time may obtain write access on an object—when that thread obtains write access, the object is locked to all other threads.

- The Two-Phase Termination design pattern ensures that a thread frees resources—such as other spawned threads—in memory (phase one) before terminating (phase two). In Java, a Thread object can use this pattern in method run. For instance, method run can contain an infinite loop that is terminated by some state change—upon termination, method run can invoke a private method responsible for stopping any other spawned threads (phase one). The thread then terminates after method run terminates (phase two). In Chapter 13, the ChatServerAdministrator and ChatServer classes of the RMI Deitel Messenger application use this design pattern, which we describe in greater detail.

### 1.5.4 Architectural Patterns

Design patterns allow developers to design specific parts of systems, such as abstracting object instantiations, aggregating classes into larger structures or assigning responsibilities to objects. Architectural patterns, on the other hand, provide developers with proven strategies for designing subsystems and specifying how they interact with each other.

For example, the Model-View-Controller architectural pattern separates application data (contained in the model) from graphical presentation components (the view) and input-processing logic (the controller). In the design for a simple text editor, the user inputs text from the keyboard and formats this text using the mouse. The program stores this text and format information into a series of data structures, then displays this information on screen for the user to read what has been inputted. The model, which contains the application data, might contain only the characters that make up the document. When a user provides some input, the controller modifies the model’s data with the given input. When the model
changes, it notifies the view of the change so the view can update its presentation with the changed data—e.g., the view might display characters using a particular font, with a particular size. Chapter 3 discusses Model-View-Controller architecture in detail, and our Java 2D drawing application in Chapter 5 and the Enterprise Java case study in Chapters 17–20 use this architecture extensively.

The Layers architectural pattern divides functionality into separate sets of system responsibilities called layers. For example, three-tier applications, in which each tier contains a unique system component, is an example of the Layers architectural pattern. This type of application contains three components that assume a unique responsibility. The information tier (also called the “bottom tier”) maintains data for the application, typically storing the data in a database. The client tier (also called the “top tier”) is the application’s user interface, such as a standard Web browser. The middle tier acts as an intermediary between the information tier and the client tier by processing client-tier requests, reading data from and writing data to the database. In this book, the three-tier architectures in the Deitel bookstore application (Chapter 11), the wireless application case study (Chapter 12) and the Enterprise Java case study (Chapters 17–20) all use the Layers architectural pattern. We discuss the nuances of each architecture in its respective chapter.

Using architectural patterns promotes extensibility when designing systems, because designers can modify a component without having to modify another. For example, a text editor that uses the Model-View-Controller architectural pattern is extensible; designers can modify the view that displays the document outline but would not have to modify the model, other views or controllers. A system designed with the Layers architectural pattern is also extensible; designers can modify the information tier to accommodate a particular database product, but they would not have to modify either the client tier or the middle tier extensively.

1.5.5 Further Study on Design Patterns

We hope that you will pursue further study of design patterns. We recommend that you visit the URLs and read the books we mention below as you study patterns throughout this book. We especially encourage you to read the gang-of-four book.

*Design Patterns*

[www.hillside.net/patterns](http://www.hillside.net/patterns)
This page has links to information on design patterns and languages.

[www.hillside.net/patterns/books/](http://www.hillside.net/patterns/books/)
This site lists books on design patterns.

[www.netobjectives.com/design.htm](http://www.netobjectives.com/design.htm)
This site overviews design patterns and motivates their importance.

[umbc7.umbc.edu/~tarr/dp/dp.html](http://umbc7.umbc.edu/~tarr/dp/dp.html)
This site links to design patterns Web sites, tutorials and papers.

[www.links2go.com/topic/Design_Patterns](http://www.links2go.com/topic/Design_Patterns)
This site links to sites and information on design patterns.

This site discusses recent advances in design patterns and ideas for future projects.
Design Patterns in Java

www.research.umbc.edu/~tarr/cs491/fall00/cs491.html
This site is for a Java design patterns course at the University of Maryland. It contains numerous examples of how to apply design patterns in Java.

www.enteract.com/~bradapp/javapats.html
This site discusses Java design patterns and presents design patterns in distributed computing.

www.meurrens.org/ip-Links/java/designPatterns/
This site displays numerous links to resources and information on Java design patterns.

Architectural Patterns

compsci.about.com/science/compsci/library/weekly/aa030600a.htm
This site provides an overview the Model-View-Controller architecture.

This site contains an article discussing how Swing components use Model-View-Controller architecture.

www.ootips.org/mvc-pattern.html
This site provides information and tips on using MVC.

www.ftech.co.uk/~honeyg/articles/pda.htm
This site includes an article on the importance of architectural patterns in software.

This site provides information about architectural patterns, design patterns and idioms (patterns targeting a specific language).

WORKS CITED
1. E. Gamma, et al, Design Patterns; Elements of Reusable Object-Oriented Software (Boston, MA: Addison-Wesley, 1995) 1–31.
2. J. Vlissides, Pattern Hatching; Design Patterns Applied (Boston, MA: Addison-Wesley, 1998) 146.

BIBLIOGRAPHY
Gamma, R., R. Helm, R. Johnson and J. Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.


Advanced Swing Graphical User Interface Components

Objectives

• To be able to use Swing components to enhance application GUIs.
• To be able to use Swing text components to view styled documents.
• To understand the Command design pattern and its implementation in Swing.
• To be able to develop applications with multiple-document interfaces.
• To understand how to implement drag-and-drop support.
• To learn how to prepare internationalized applications.
• To understand how to use Swing to create accessible applications for people with disabilities.

The best investment is in the tools of one’s own trade.
Benjamin Franklin

Every action must be due to one or other of seven causes: chance, nature, compulsion, habit, reasoning, anger or appetite.
Aristotle

Happiness, like an old friend, is inclined to drop in unexpectedly—when you are working hard on something else.
Ray Inman
### 2.1 Introduction

In this chapter, we introduce Swing components that enable developers to build functionally rich user interfaces. The Swing graphical user interface components were introduced with the Java Foundation Classes (JFC) as a downloadable extension to the Java 1.1 Platform, then became a standard extension in the Java 2 Platform. Swing provides a more complete set of GUI components than the Abstract Windowing Toolkit (AWT), including advanced features such as a pluggable look and feel, lightweight component rendering and drag-and-drop capabilities.

We introduce the `JEditorPane` class for rendering styled content, such as HTML pages, and build a simple Web browser. We continue our discussion of design patterns by introducing Swing `Actions`, which implement the Command design pattern. Swing `Actions` enable developers to build reusable, user-interface logic components. We also introduce `JSplitPane`, `JTabbedPane` and multiple-document-interface components for organizing GUI elements.

Java provides mechanisms for building applications for multiple languages and countries, and for disabled users. Building internationalized applications ensures that applications will be ready for use around the world in many languages and countries. Accessibility ensures that disabled users will be able to use applications through commonly available utilities, such as screen readers. We show how Swing components enable Java developers to build applications that are accessible to users with disabilities.

### 2.2 WebBrowser Using JEditorPane and JToolBar

In this section, we use Swing components to build a simple Web browser. We introduce Swing’s advanced text-rendering capabilities and containers for grouping commonly used interface elements for convenient user access.
2.2.1 Swing Text Components and HTML Rendering

Many applications present text to the user for viewing and editing. This text may consist of plain, unformatted characters, or it may consist of richly styled characters that use multiple fonts and extensive formatting. Swing provides three basic types of text components for presenting and editing text. Class \texttt{JTextComponent} is the base class for all Swing text components, including \texttt{JTextField}, \texttt{JTextArea} and \texttt{JEditorPane}.

\texttt{JTextField} is a single-line text component suitable for obtaining simple user input or displaying information such as form field values, calculation results and so on. \texttt{JPasswordField} is a subclass of \texttt{JTextField} suitable for obtaining user passwords. These components do not perform any special text styling. Rather, they present all text in a single font and color. \texttt{JTextArea}, like \texttt{JTextField} and \texttt{JPasswordField}, also does not style its text. However, \texttt{JTextArea} does provide a larger visible area and supports larger plain-text documents.

\texttt{JEditorPane} provides enhanced text-rendering capabilities. \texttt{JEditorPane} supports styled documents that include formatting, font and color information. \texttt{JEditorPane} is capable of rendering HTML documents as well as Rich Text Format (RTF) documents. We use class \texttt{JEditorPane} to render HTML pages for a simple Web-browser application. \texttt{JTextPane} is a \texttt{JEditorPane} subclass that renders only styled documents, and not plain text. \texttt{JTextPane} provides developers with fine-grained control over the style of each character and paragraph in the rendered document.

\texttt{WebBrowserPane} (Fig. 2.1) extends class \texttt{JEditorPane} to create a Web-browsing component that maintains a history of visited URLs. Line 16 creates a \texttt{List} for keeping track of visited URLs. Line 23 invokes method \texttt{setEditable} of class \texttt{JEditorPane} to disable text editing in the \texttt{WebBrowserPane}. \texttt{JEditorPane} enables hyperlinks in HTML documents only if the \texttt{JEditorPane} is not editable.

```java
// WebBrowserPane.java
// WebBrowserPane is a simple Web-browsing component that
// extends JEditorPane and maintains a history of visited URLs.
package com.deitel.advjhtpl.gui.webbrowser;

// Java core packages
import java.util.*;
import java.net.*;
import java.io.*;

// Java extension packages
import javax.swing.*;

public class WebBrowserPane extends JEditorPane {
	private List history = new ArrayList();
	private int historyIndex;
}```

Fig. 2.1 WebBrowserPane subclass of \texttt{JEditorPane} for viewing Web sites and maintaining URL history (part 1 of 3).
// WebBrowserPane constructor
public WebBrowserPane()
{
    // disable editing to enable hyperlinks
    setEditable( false );
}

// display given URL and add it to history
public void goToURL( URL url )
{
    displayPage( url );
    history.add( url );
    historyIndex = history.size() - 1;
}

// display next history URL in editorPane
public URL forward()
{
    historyIndex++;
    // do not go past end of history
    if ( historyIndex >= history.size() )
        historyIndex = history.size() - 1;
    URL url = ( URL ) history.get( historyIndex );
    displayPage( url );
    return url;
}

// display previous history URL in editorPane
public URL back()
{
    historyIndex--;
    // do not go past beginning of history
    if ( historyIndex < 0 )
        historyIndex = 0;
    // display previous URL
    URL url = ( URL ) history.get( historyIndex );
    displayPage( url );
    return url;
}

// display given URL in JEditorPane
private void displayPage( URL pageURL )
{

Fig. 2.1 WebBrowserPane subclass of JEditorPane for viewing Web sites and maintaining URL history (part 2 of 3).
Method `goToURL` (lines 27–32) navigates the `WebBrowserPane` to the given URL. Line 29 invokes method `displayPage` of class `WebBrowserPane` to display the given URL. Line 30 invokes method `add` of interface `List` to add the URL to the browser history. Line 31 updates the `historyIndex` to ensure that methods `back` and `forward` navigate to the appropriate URL.

Method `forward` (lines 35–47) navigates the `WebBrowserPane` to the next page in the URL history. Line 37 increments `historyIndex`, and lines 43–44 retrieve the URL from the `historyList` and display the URL in `WebBrowserPane`. If the `historyIndex` is past the last page in the `history`, line 41 sets `historyIndex` to the last URL in `history`.

Method `back` (lines 50–63) navigates `WebBrowserPane` to the previous page in the URL history. Line 52 decrements `historyIndex`, and lines 55–56 ensure that `historyIndex` does not fall below 0. Lines 59–60 retrieve the URL and display it in the `WebBrowserPane`.

Method `displayPage` takes as an argument a `URL` to display in the `WebBrowserPane`. Line 70 invokes method `setPage` of class `JEditorPane` to display the page that the URL references. Lines 74–76 catch an `IOException` if there is an error loading the page from the given `URL`.

### 2.2.2 Swing Toolbars

Toolbars are GUI containers typically located below an application’s menus. Toolbars contain buttons and other GUI components for commonly used features, such as cut, copy and paste, or navigation buttons for a Web browser. Figure 2.2 shows toolbars in Internet Explorer and Mozilla.

Class `javax.swing.JToolBar` enables developers to add toolbars to Swing user interfaces. `JToolBar` also enables users to modify the appearance of the `JToolBar` in a running application. For example, the user can drag the `JToolBar` from the top of a window and "dock" the `JToolBar` on the side or bottom of the window. Users also can drag the `JToolBar` away from the application window (Fig. 2.4) to create a `floating JToolBar` (i.e., a `JToolBar` displayed in its own window). Developers can set `JToolBar` properties that enable or disable dragging and floating.
WebToolBar (Fig. 2.3) extends class JToolBar to provide commonly used navigation components for a WebBrowserPane. WebToolBar provides backButton (line 20) for navigating to the previous page, forwardButton (line 21) for navigating to the next page and urlTextField to allow the user to enter a URL (line 22).

The WebToolBar constructor (lines 25–96) takes as an argument a WebBrowserPane for displaying Web pages. Lines 34–53 create urlTextField and its associated ActionListener. When a user types a URL and hits the Enter key, line 44 invokes method goToURL of class WebBrowserPane to display the user-entered URL.

Lines 56–57 create backButton, which allows the user to navigate to the previously viewed Web site. Recall that class WebBrowserPane maintains a history of visited URLs. When the user selects backButton, line 65 invokes method back of class Web-
BrowserPane to navigate to the previous URL. Method back returns the destination URL, which line 68 displays in urlTextField. This ensures that urlTextField shows the proper URL for the Web site displayed in the WebBrowserPane.

Lines 74–75 create forwardButton, which allows the user to navigate forward through the WebBrowserPane’s history of visited URLs. When the user activates forwardButton, line 83 invokes method forward of class WebBrowserPane to navigate to the next URL in the WebBrowserPane’s URL history. Line 86 displays the URL in urlTextField.

```
// WebToolBar.java
// WebToolBar is a JToolBar subclass that contains components
// for navigating a WebBrowserPane. WebToolBar includes back
// and forward buttons and a text field for entering URLs.
package com.deitel.advjhtp1.gui.webbrowser;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.net.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

public class WebToolBar extends JToolBar
    implements HyperlinkListener {
private WebBrowserPane webBrowserPane;
private JButton backButton;
private JButton forwardButton;
private JTextField urlTextField;

    // WebToolBar constructor
    public WebToolBar( WebBrowserPane browser )
    {
        super( "Web Navigation" );

        // register for HyperlinkEvents
        webBrowserPane = browser;
        webBrowserPane.addHyperlinkListener( this );

        // create JTextField for entering URLs
        urlTextField = new JTextField( 25 );
        urlTextField.addActionListener(
            new ActionListener() {
                }
            });

Fig. 2.3  WebToolBar JToolBar subclass for navigating URLs in a
WebBrowserPane (part 1 of 3).
```
// navigate webBrowser to user-entered URL
public void actionPerformed( ActionEvent event )
{
    // attempt to load URL in webBrowserPane
    try {
        URL url = new URL( urlTextField.getText() );
        webBrowserPane.goToURL( url );
    }

    // handle invalid URL
    catch ( MalformedURLException urlException ) {
        urlException.printStackTrace();
    }
}

// create JButton for navigating to previous history URL
backButton = new JButton( new ImageIcon( getClass().getResource( "images/back.gif" ) ) );
backButton.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent event )
    {
        // navigate to previous URL
        URL url = webBrowserPane.back();

        // display URL in urlTextField
        urlTextField.setText( url.toString() );
    }
});

// create JButton for navigating to next history URL
forwardButton = new JButton( new ImageIcon( getClass().getResource( "images/forward.gif" ) ) );
forwardButton.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent event )
    {
        // navigate to next URL
        URL url = webBrowserPane.forward();

        // display new URL in urlTextField
        urlTextField.setText( url.toString() );
    }
});

Fig. 2.3  WebToolBar JToolBar subclass for navigating URLs in a WebBrowserPane (part 2 of 3).
Based on class JToolBar’s inheritance hierarchy, each JToolBar also is a java.awt.Container and therefore can contain other GUI components. Lines 92–94 add backButton, forwardButton and urlTextField to the WebToolBar by invoking method add of class JToolBar. A JToolBar has property orientation that specifies how the JToolBar will arrange its child components. The default is horizontal orientation, so the JToolBar lays out these components next to one another, left to right.

Class WebBrowserPane renders HTML pages, which may contain hyperlinks to other Web pages. When a user activates a hyperlink in a WebBrowserPane (e.g., by clicking on the hyperlink), the WebBrowserPane issues a HyperlinkEvent of type HyperlinkEvent.EventTypes.ACTIVATED. Class WebToolBar implements interface HyperlinkListener to listen for HyperlinkEvents. There are several HyperlinkEvent types. Method hyperlinkUpdate (lines 99–112) invokes method getEventType of class HyperlinkEvent to check the event type (lines 102–103) and retrieves the HyperlinkEvent’s URL (line 106). This is the URL of the user-selected hyperlink. Line 109 invokes method goToURL of class WebBrowserPane to navigate to the selected URL, and line 110 updates urlTextField to display the selected URL.
Class **WebBrowser** (Fig. 2.4) uses a **WebBrowserPane** and **WebToolBar** to create a simple Web-browser application. Line 26 creates a **WebBrowserPane**, and line 27 creates a **WebToolBar** for this **WebBrowserPane**. Lines 31–33 add the **WebBrowserPane** and **WebToolBar** to the **WebBrowser**’s content pane.

```java
package com.deitel.advjhtp1.gui.webbrowser;

import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;
import javax.swing.event.*;

public class WebBrowser extends JFrame {

    private WebToolBar toolBar;
    private WebBrowserPane browserPane;

    // WebBrowser constructor
    public WebBrowser() {
        super( "Deitel Web Browser" );

        // create WebBrowserPane and WebToolBar for navigation
        browserPane = new WebBrowserPane();
        toolBar = new WebToolBar( browserPane );

        // lay out WebBrowser components
        Container contentPane = getContentPane();
        contentPane.add( toolBar, BorderLayout.NORTH );
        contentPane.add( new JScrollPane( browserPane ),
                            BorderLayout.CENTER );
    }

    // execute application
    public static void main( String args[] ) {
        WebBrowser browser = new WebBrowser();
        browser.setDefaultCloseOperation( EXIT_ON_CLOSE );
        browser.setSize( 640, 480 );
        browser.setVisible( true );
    }
}
```

**Fig. 2.4** **WebBrowser** application for browsing Web sites using **WebBrowserPane** and **WebToolBar** (part 1 of 2).

2.3 Swing Actions

Applications often provide users with several different ways to perform a given task. For example, in a word processor there might be an Edit menu with menu items for cutting, copying and pasting text. There also might be a toolbar that has buttons for cutting, copying and pasting text. There also might be a pop-up menu to allow users to right click on a document to cut, copy or paste text. The functionality the application provides is the same in each case—the developer provides the various interface components for the user’s convenience. However, the same GUI component instance (e.g., a JButton for cutting text)
cannot be used for menus and toolbars and pop-up menus, so the developer must code the same functionality three times. If there were many such interface items, repeating this functionality would become tedious and error-prone.

The *Command design pattern* solves this problem by enabling developers to define the functionality (e.g., copying text) once in a reusable object that the developer then can add to a menu, toolbar or pop-up menu. This design pattern is called Command because it defines a user command or instruction. The *Action* interface defines required methods for the Java Swing implementation of the Command design pattern.

An *Action* represents user-interface logic and properties for GUI components that represent that logic, such as the label for a button, the text for a tool tip and the mnemonic key for keyboard access. The logic takes the form of an `actionPerformed` method that the event mechanism invokes in response to the user activating an interface component (e.g., the user clicking a *JButton*). Interface *Action* extends interface *ActionListener*, which enables *Actions* to process *ActionEvents* generated by GUI components. Once a developer defines an *Action*, the developer can add that *Action* to a *JMenu* or *JToolBar*, just as if the *Action* were a *JMenuItem* or *JButton*. For example, when a developer adds an *Action* to a *JMenu*, the *JMenu* creates a *JMenuItem* for the *Action* and uses the *Action* properties to configure the *JMenuItem*.

*Actions* provide an additional benefit in that the developer can enable or disable all GUI components associated with an *Action* by enabling or disabling the *Action* itself. For example, copying text from a document first requires that the user select the text to be copied. If there is no selected text, the program should not allow the user to perform a copy operation. If the application used a separate *JMenuItem* in a *JMenu* and *JButton* in a *JToolBar* for copying text, the developer would need to disable each of these GUI components individually. Using *Actions*, the developer could disable the *Action* for copying text, which also would disable all associated GUI components.

*ActionSample* (Fig. 2.5) demonstrates two *Actions*. Lines 15–16 declare *Action* references `sampleAction` and `exitAction`.

```java
// ActionSample.java
// Demonstrating the Command design pattern with Swing Actions.
package com.deitel.advjhtp1.gui.actions;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class ActionSample extends JFrame {
    // Swing Actions
    private Action sampleAction;
    private Action exitAction;

    // Application class
    public static void main(String[] args) {
        ActionSample frame = new ActionSample();
        frame.setVisible(true);
    }
}
```

*Fig. 2.5* *ActionSample* application demonstrating the Command design pattern with Swing *Actions* (part 1 of 4).
// ActionSample constructor
public ActionSample()
{
    super( "Using Actions" );

    // create AbstractAction subclass for sampleAction
    sampleAction = new AbstractAction() {
        public void actionPerformed( ActionEvent event )
        {
            // display message indicating sampleAction invoked
            JOptionPane.showMessageDialog( ActionSample.this,
            "The sampleAction was invoked" );

            // enable exitAction and associated GUI components
            exitAction.setEnabled( true );
        }
    };

    // set Action name
    sampleAction.putValue( Action.NAME, "Sample Action" );

    // set Action Icon
    sampleAction.putValue( Action.SMALL_ICON, new ImageIcon( getClass().getResource( "images/Help24.gif" ) ) );

    // set Action short description (tooltip text)
    sampleAction.putValue( Action.SHORT_DESCRIPTION,
    "A Sample Action" );

    // set Action mnemonic key
    sampleAction.putValue( Action.MNEMONIC_KEY,
    new Integer( 'S' ) );

    // create AbstractAction subclass for exitAction
    exitAction = new AbstractAction() {
        public void actionPerformed( ActionEvent event )
        {
            // display message indicating exitAction invoked
            JOptionPane.showMessageDialog( ActionSample.this,
            "The exitAction was invoked" );
            System.exit( 0 );
        }
    };

    // set Action name
    exitAction.putValue( Action.NAME, "Exit" );

    // set Action icon
    exitAction.putValue( Action.SMALL_ICON, new ImageIcon( getClass().getResource( "images/EXIT.gif" ) ) );

Fig. 2.5 ActionSample application demonstrating the Command design pattern with Swing Actions (part 2 of 4).
// set Action short description (tooltip text)
exitAction.putValue( Action.SHORT_DESCRIPTION, "Exit Application" );

// set Action mnemonic key
exitAction.putValue( Action.MNEMONIC_KEY, new Integer( 'x' ) );

// disable exitAction and associated GUI components
exitAction.setEnabled( false );

// create File menu
JMenu fileMenu = new JMenu( "File" );

// add sampleAction and exitAction to File menu to create a JMenuItem for each Action
fileMenu.add( sampleAction );
fileMenu.add( exitAction );

fileMenu.setMnemonic( 'F' );

// create JMenuBar and add File menu
JMenuBar menuBar = new JMenuBar();
menuBar.add( fileMenu );
setJMenuBar( menuBar );

// create JToolBar
JToolBar toolBar = new JToolBar();

// add sampleAction and exitAction to JToolBar to create JButtons for each Action
toolBar.add( sampleAction );
toolBar.add( exitAction );

// create JButton and set its Action to sampleAction
JButton sampleButton = new JButton();
sampleButton.setAction( sampleAction );

// create JButton and set its Action to exitAction
JButton exitButton = new JButton( exitAction );

// lay out JButtons in JPanel
JPanel buttonPanel = new JPanel();
buttonPanel.add( sampleButton );
buttonPanel.add( exitButton );

// add toolBar and buttonPanel to JFrame's content pane
Container container = getContentPane();
container.add( toolBar, BorderLayout.NORTH );
container.add( buttonPanel, BorderLayout.CENTER );
}

Fig. 2.5 ActionSample application demonstrating the Command design pattern with Swing Actions (part 3 of 4).
Lines 24–35 create an anonymous inner class that extends class `AbstractAction` and assigns the instance to reference `sampleAction`. Class `AbstractAction` facilitates creating `Action` objects. Class `AbstractAction` implements interface `Action`, but is marked `abstract` because class `AbstractAction` does not provide an implementation for method `actionPerformed`. Lines 26–34 implement method `actionPerformed`. The Swing event mechanism invokes method `actionPerformed` when the user activates a GUI component associated with `sampleAction`. We show how to create these GUI components shortly. Lines 29–30 in method `actionPerformed` display a `JOptionPane` message dialog to inform the user that `sampleAction` was invoked. Line 33 then invokes method `setEnabled` of interface `Action` on the `exitAction` reference. This enables the `exitAction` and its associated GUI components. Note that `Actions` are enabled by default. We disabled the `exitAction` (line 80) to demonstrate that this disables the GUI components associated with that `Action`.

After instantiating an `AbstractAction` subclass to create `sampleAction`, lines 38–50 repeatedly invoke method `putValue` of interface `Action` to configure `sampleAction` properties. Each property has a key and a value. Interface `Action` defines the keys as `public` constants, which we list in Fig. 2.6. GUI components associated with `sampleAction` use the property values we assign for GUI component labels, icons, tooltips and so on. Line 38 invokes method `putValue` of interface `Action` with arguments `Action.NAME` and "Sample Action". This assigns `sampleAction`'s name, which GUI components use as their label. Lines 41–42 invoke method `putValue` of interface `Action` with key `Action.SMALL_ICON` and an `ImageIcon` value, which GUI components use as their `Icon`. Lines 45–46 set the `Action`'s tool tip using key
Lines 49–50 set the Action’s mnemonic key using key Action.MNEMONIC_KEY. When the Action is placed in a JMenu, the mnemonic key provides keyboard access to the Action. Lines 53–80 create the exitAction in a similar way to sampleAction, with an appropriate name, icon, description and mnemonic key. Line 80 invokes method setEnabled of interface Action with argument false to disable the exitAction. We use this to demonstrate that disabling an Action also disables the Action’s associated GUI components.

Line 83 creates the fileMenu JMenu, which contains JMenuItems corresponding to sampleAction and exitAction. Class JMenu overloads method add with a version that takes an Action argument. This overloaded add method returns a reference to the JMenuItem that it creates. Lines 87–88 invoke method add of class JMenuItem to add sampleAction and exitAction to the menu. We have no need for the JMenuItem references that method add returns, so we ignore them. Line 90 sets the fileMenu mnemonic key, and lines 93–95 add the fileMenu to a JMenuBar and invoke method setJMenuBar of class JFrame to add the JMenuBar to the application.

Lines 98 creates a new JToolBar. Like JMenu, JToolBar also provides overloaded method add for adding Actions to JToolBars. Method add of class JToolBar returns a reference to the JButton created for the given Action. Lines 102–103 invoke method add of class JToolBar to add the sampleAction and exitAction to the JToolBar. We have no need for the JButton references that method add returns, so we ignore them.


Software Engineering Observation 2.1

According to the Java 2 SDK documentation, it is preferable to create JButtons and JMenuItem, invoke method setAction then add the JButton or JMenuItem to its container, rather than adding the Action to the container directly. This is because most GUI-building tools do not support adding Actions to containers directly.

Lines 113–120 add the newly created JButtons to a JPanel and lay out the JToolBar and JPanel in the JFrame’s content pane. Note that in the first screen capture of Fig. 2.5, the JButtons for exitAction appear grayed-out. This is because the exitAction is disabled. After invoking the sampleAction, the exitAction is enabled and appears in full color. Note also the tool tips, icons and labels on each GUI component. Each of these items was configured using properties of the respective Action object.

Figure 2.6 summarizes Action properties. Each property name is a static constant in interface Action and acts as a key for setting or retrieving the property value.

In the following sections we demonstrate two alternative ways to create Swing Action instances. The first uses named inner classes. The second defines a generic AbstractAction subclass that provides a constructor for commonly used properties and set methods for each individual Action property.
2.4 **JSplitPane and JTabbedPane**

*JSplitPane* and *JTabbedPane* are container components that enable developers to present large amounts of information in a small screen area. *JSplitPane* accomplishes this by dividing two components with a divider users can reposition to expand and contract the visible areas of the *JSplitPane*’s child components (Fig. 2.7). *JTabbedPane* uses a file-folder-style tab interface to arrange many components through which the user can browse.

**Look-and-Feel Observation 2.1**

*JSplitPane* can contain only two child components. However, each child component may contain nested components.

*FavoritesWebBrowser* (Fig. 2.7) is an application that uses a *JSplitPane* to show two *WebBrowserPane* components side-by-side in a single application window. On the left side, the *JSplitPane* contains a *WebBrowserPane* that displays a static HTML page containing links to the user’s favorite Web sites. Activating the links in this favorites page displays the URL contents in the *WebBrowserPane* on the right side of the *JSplitPane*. This is a common user interface arrangement in Web browsers, such as Internet Explorer and Netscape Navigator.

```java
// FavoritesWebBrowser.java
// FavoritesWebBrowser is an application for browsing Web sites
// using a WebToolBar and WebBrowserPane and displaying an HTML
// page containing links to favorite Web sites.
package com.deitel.advjhtp1.gui.splitpane;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.net.*;
```
// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

// Deitel packages
import com.deitel.advjhtp1.gui.webbrowser.*;

public class FavoritesWebBrowser extends JFrame {
    private WebToolBar toolBar;
    private WebBrowserPane browserPane;
    private WebBrowserPane favoritesBrowserPane;

    // WebBrowser constructor
    public FavoritesWebBrowser() {
        super( "Deitel Web Browser" );

        // create WebBrowserPane and WebToolBar for navigation
        browserPane = new WebBrowserPane();
        toolBar = new WebToolBar( browserPane );

        // create WebBrowserPane for displaying favorite sites
        favoritesBrowserPane = new WebBrowserPane();

        // add WebToolBar as listener for HyperlinkEvents
        favoritesBrowserPane.addHyperlinkListener( toolBar );

        // display favorites.html in favoritesBrowserPane
        favoritesBrowserPane.goToURL( getClass().getResource( "favorites.html" ) );

        // create JSplitPane with horizontal split (side-by-side)
        JSplitPane splitPane = new JSplitPane( JSplitPane.HORIZONTAL_SPLIT,
                                               new JScrollPane( favoritesBrowserPane ),
                                               new JScrollPane( browserPane ) );

        // position divider between WebBrowserPanes
        splitPane.setDividerLocation( 210 );

        // add buttons for expanding/contracting divider
        splitPane.setOneTouchExpandable( true );

        // lay out WebBrowser components
        Container contentPane = getContentPane();
        contentPane.add( toolBar, BorderLayout.NORTH );
        contentPane.add( splitPane, BorderLayout.CENTER );
    }
}

Fig. 2.7 FavoritesWebBrowser application for displaying two Web pages side-by-side using JSplitPane (part 2 of 3).
Lines 31–32 create a `WebBrowserPane` for displaying Web pages and a `WebToolBar` for navigating this `WebBrowserPane`. Line 35 creates an additional `WebBrowser` pane called `favoritesBrowserPane`, which the application will use to display `favorites.html`. This HTML document contains hyperlinks to some favorite Web sites. Line 39 invokes method `addHyperlinkListener` of class `WebBrowser-`
Pane to register the toolBar as a HyperlinkListener for favorites-BrowserPane. When a user activates a link in favoritesBrowserPane, toolBar will receive the HyperlinkEvent and display the activated URL in browserPane. This way the user can activate links in favoritesBrowserPane and display those links in browserPane. Lines 42–43 invoke method goToURL of class WebBrowserPane to load favorites.html in favoritesBrowserPane.

Lines 47–50 create a JSplitPane. This JSplitPane constructor takes as its first argument an integer that indicates the JSplitPane orientation. The constant JSplitPane.HORIZONTAL_SPLIT specifies the JSplitPane should display its child components side-by-side. The constant JSplitPane.VERTICAL_SPLIT would specify that the JSplitPane should display its child components one on top of the other. The second and third arguments to this JSplitPane constructor are the components to be divided in the JSplitPane. In this case, we add favoritesBrowserPane to the left side of the JSplitPane and browserPane to the right side of the JSplitPane. We place each WebBrowserPane in a JScrollPane to allow the user to scroll if the content exceeds the visible area.

Line 53 invokes method setDividerLocation of class JSplitPane to set the exact divider position between favoritesBrowserPane and browserPane. Line 56 invokes method setOneTouchExpandable of class JSplitPane to add two buttons to the divider that enable the user to expand or collapse the divider to one side or the other with a single click. Note the arrows on the divider in Fig. 2.7.

**Good Programming Practice 2.1**

Place child components in JScrollPanes before adding the components to a JSplitPane. This ensures that the user will be able to view all the content in each child component by scrolling if necessary.

JTabbedPane presents multiple components in separate tabs, which the user can navigate using a mouse or keyboard. Dialog boxes often use components similar to JTabbedPane. For example, Fig. 2.8 shows the Display Properties tabbed dialog in Windows 2000.

![Fig. 2.8 Tabbed interface of Display Properties dialog box in Windows 2000.](image)
TabbedPaneWebBrowser (Fig. 2.9) uses a JTabbedPane to enable users to browse multiple Web pages at once in a single application window. The user invokes an Action to add a new WebBrowserPane to the JTabbedPane. Each time the user adds a new WebBrowserPane, the JTabbedPane creates a new tab and places the WebBrowserPane in this new tab.
private void createNewTab()
{
    // create JPanel to contain WebBrowserPane and WebToolBar
    JPanel panel = new JPanel( new BorderLayout() );
    WebBrowserPane browserPane = new WebBrowserPane();
    WebToolBar toolBar = new WebToolBar( browserPane );

    // add WebBrowserPane and WebToolBar to JPanel
    panel.add( toolBar, BorderLayout.NORTH );
    panel.add( new JScrollPane( browserPane ),
               BorderLayout.CENTER );

    // add JPanel to JTabbedPane
    tabbedPane.addTab( "Browser " + tabbedPane.getTabCount(),
                      panel );
}

public NewTabAction()
{
    // set name, description and mnemonic key
    putValue( Action.NAME, "New Browser Tab" );
    putValue( Action.SHORT_DESCRIPTION,
              "Create New Web Browser Tab" );
    putValue( Action.MNEMONIC_KEY, new Integer( 'N' ) );
}

public void actionPerformed( ActionEvent event )
{
    createNewTab();
}

private class ExitAction extends AbstractAction {

    // ExitAction constructor
    public ExitAction()
    {
        // set name, description and mnemonic key
        putValue( Action.NAME, "Exit" );
        putValue( Action.SHORT_DESCRIPTION, "Exit Application" );
        putValue( Action.MNEMONIC_KEY, new Integer( 'x' ) );
    }

    public void actionPerformed( ActionEvent event )
    {
        System.exit( 0 );
    }
}

// Action for exiting application
private class ExitAction extends AbstractAction {

    // ExitAction constructor
    public ExitAction()
    {
        // set name, description and mnemonic key
        putValue( Action.NAME, "Exit" );
        putValue( Action.SHORT_DESCRIPTION, "Exit Application" );
        putValue( Action.MNEMONIC_KEY, new Integer( 'x' ) );
    }

    public void actionPerformed( ActionEvent event )
    {
        System.exit( 0 );
    }

    // Action for exiting application
    private class ExitAction extends AbstractAction {

        // ExitAction constructor
        public ExitAction()
        {
            // set name, description and mnemonic key
            putValue( Action.NAME, "Exit" );
            putValue( Action.SHORT_DESCRIPTION, "Exit Application" );
            putValue( Action.MNEMONIC_KEY, new Integer( 'x' ) );
        }

        public void actionPerformed( ActionEvent event )
        {
            System.exit( 0 );
        }
}

TabbedPaneWebBrowser application using JTabbedPane to
browse multiple Web sites concurrently (part 2 of 3).
Line 19 creates a new `JTabbedPane`, to which the user will add `WebBrowserPanels`. Line 27 invokes method `createNewTab` of class `TabbedPaneWebBrowser` to create the first `WebBrowserPane` and place it in the `JTabbedPane`. Line 30 adds the `JTabbedPane` to the `TabbedPaneWebBrowser`’s content pane. Lines 34–42 create the `File` menu, which contains an `Action` for creating new `WebBrowserPanels` (line 35) and an `Action` for exiting the application (line 37). We discuss these actions in detail momentarily.

Method `createNewTab` (lines 46–64) creates a new `WebBrowserPane` and adds it to the `JTabbedPane`. Line 50 creates a `JPanel` for laying out the `WebBrowserPane` and its `WebToolBar`. Lines 53–59 create a `WebBrowserPane` and a `WebToolBar` and add them to the `JPanel`. Lines 62–63 invoke method `addTab` of class `JTabbedPane` to add the `JPanel` containing the `WebBrowserPane` and `WebToolBar` to the application’s `JTabbedPane`. Method `addTab` of class `JTabbedPane` takes as a `String` argument the title for the new tab and as a `Component` argument the `Component` to display in the new tab. Although a developer may add any `Component` instance to a `JTabbedPane` to create a new tab, developers most commonly lay out components in a `JPanel` and add the `JPanel` to the `JTabbedPane`.

Figure 2.9 also demonstrates a second way to create `Action` instances. Lines 67–84 define inner class `NewTabAction`, which extends `AbstractAction`. The `NewTabAction` constructor (lines 70–77) configures the `Action` by invoking method `putValue` for the `Action` name, tool tip and mnemonic key. Lines 80–83 define method `actionPerformed` and invoke method `createNewTab` (line 82) to create a new tab in the `JTabbedPane` containing a `WebBrowserPane` and `WebToolBar`.

Lines 87–103 define inner class `ExitAction`, which also extends `AbstractAction`. The `ExitAction` constructor (lines 90–96) configures the `Action` name, tool tip and mnemonic key by invoking method `putValue`. Method `actionPerformed` (lines 99–102) invokes `static` method `exit` of class `System` to exit the application.
2.5 Multiple-Document Interfaces

Most applications provide a single-document interface—users can view and edit only one document at a time. For example, most Web browsers allow users to view only one Web page. To view multiple Web pages, users must launch additional Web browsers. *Multiple document interfaces* allow users to view multiple documents in a single application. Each document appears in a separate window in the application. The user can arrange, resize, iconify (i.e., minimize) and maximize these separate document windows like application windows on the desktop. For example, a digital photograph-editing application could use a multiple document interface to enable users to view and edit multiple photographs at once. The user could place the photograph windows side-by-side to compare the photographs or copy part of one photograph and paste it into the other.

Java Swing provides classes *JDesktopPane* and *JInternalFrame* for building multiple-document interfaces. These class names reinforce the idea that each document is a separate window (*JInternalFrame*) inside the application’s desktop (*JDesktopPane*), just as other applications are separate windows (e.g., *JFrames*) on the operating system’s desktop. *JInternalFrames* behave much like *JFrames*. Users can maximize, iconify, resize, open and close *JInternalFrames*. *JInternalFrames* have title bars with buttons for iconifying, maximizing and closing. Users also can move *JInternalFrames* within the *JDesktopPane*.

`MDIWebBrowser` (Fig. 2.10) uses *JInternalFrames* and a *JDesktopPane* to enable users to browse multiple Web sites within a single application window. Line 20 creates a *JDesktopPane*, which is a container for *JInternalFrames*. Line 32 adds the *JDesktopPane* to the *JFrame*’s content pane. Lines 36–44 construct the application menu. The *File* menu includes an *Action* for creating new browser windows (line 37) and an *Action* for exiting the application (line 39).

```java
// MDIWebBrowser.java
// MDIWebBrowser is an application that uses JDesktopPane
// and JInternalFrames to create a multiple-document-interface
// application for Web browsing.
package com.deitel.advjhtp1.gui.mdi;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.gui.webbrowser.*;

public class MDIWebBrowser extends JFrame {

    // JDesktopPane for multiple document interface
    JDesktopPane desktopPane = new JDesktopPane();
```

*Fig. 2.10*  *MDIWebBrowser* application using *JDesktopPane* and *JInternalFrames* to browse multiple Web sites concurrently (part 1 of 4).
// MDIWebBrowser constructor
public MDIWebBrowser()
{
    super( "MDI Web Browser" );

    // create first browser window
    createNewWindow();

    // add JDesktopPane to contentPane
    Container contentPane = getContentPane();
    contentPane.add( desktopPane );

    // create File JMenu for creating new windows and exiting application
    JMenu fileMenu = new JMenu( "File" );
    fileMenu.add( new NewWindowAction() );
    fileMenu.addSeparator();
    fileMenu.add( new ExitAction() );
    fileMenu.setMnemonic( 'F' );

    JMenuBar menuBar = new JMenuBar();
    menuBar.add( fileMenu );
    setJMenuBar( menuBar );
}

// create new browser window
private void createNewWindow()
{
    // create new JInternalFrame that is resizable, closable, maximizable and iconifiable
    JInternalFrame frame = new JInternalFrame( "Browser", // title
                                               true,       // resizable
                                               true,       // closable
                                               true,       // maximizable
                                               true );     // iconifiable

    // create WebBrowserPane and WebToolBar
    WebBrowserPane browserPane = new WebBrowserPane();
    WebToolBar toolBar = new WebToolBar( browserPane );

    // add WebBrowserPane and WebToolBar to JInternalFrame
    Container contentPane = frame.getContentPane();
    contentPane.add( toolBar, BorderLayout.NORTH );
    contentPane.add( new JScrollPane( browserPane ),
                     BorderLayout.CENTER );

    // make JInternalFrame opaque and set its size
    frame.setSize( 320, 240 );

Fig. 2.10 MDIWebBrowser application using JDesktopPane and JInternalFrames to browse multiple Web sites concurrently (part 2 of 4).
// move JInternalFrame to prevent it from obscuring others
int offset = 30 * desktopPane.getAllFrames().length;
frame.setLocation( offset, offset );

// add JInternalFrame to JDesktopPane
desktopPane.add( frame );

// make JInternalFrame visible
frame.setVisible( true );
}

// Action for creating new browser windows
private class NewWindowAction extends AbstractAction {

  // NewWindowAction constructor
  public NewWindowAction() {
    // set name, description and mnemonic key
    putValue( Action.NAME, "New Window" );
    putValue( Action.SHORT_DESCRIPTION, "Create New Web Browser Window" );
    putValue( Action.MNEMONIC_KEY, new Integer( 'N' ) );
  }

  // when Action invoked, create new browser window
  public void actionPerformed( ActionEvent event ) {
    createNewWindow();
  }
}

// Action for exiting application
private class ExitAction extends AbstractAction {

  // ExitAction constructor
  public ExitAction() {
    // set name, description and mnemonic key
    putValue( Action.NAME, "Exit" );
    putValue( Action.SHORT_DESCRIPTION, "Exit Application" );
    putValue( Action.MNEMONIC_KEY, new Integer( 'x' ) );
  }

  // when Action invoked, exit application
  public void actionPerformed( ActionEvent event ) {
    System.exit( 0 );
  }
}

Fig. 2.10 MDIWebBrowser application using JDesktopPane and JInternalFrames to browse multiple Web sites concurrently (part 3 of 4).
```java
// execute application
public static void main( String args[] )
{
    MDIWebBrowser browser = new MDIWebBrowser();
    browser.setDefaultCloseOperation( EXIT_ON_CLOSE );
    browser.setSize( 640, 480 );
    browser.setVisible( true );
}
```

![Diagram of MDIWebBrowser application using JDesktopPane and JInternalFrames to browse multiple Web sites concurrently (part 4 of 4).]
Method `createNewWindow` (lines 48–81) creates a new `JInternalFrame` in response to the user invoking `NewWindowAction`. Lines 52–57 create a new `JInternalFrame` with the title "Browser". The four `boolean` arguments to the `JInternalFrame` constructor specify that the `JInternalFrame` is resizable, closable, maximizable and iconifiable. Lines 60–61 create a `WebBrowserPane` and `WebToolBar` for displaying and navigating Web pages. Like a `JFrame`, a `JInternalFrame` has a content pane. Line 64 invokes method ` getContentPane` to get the `JInternalFrame`’s content pane, and lines 65–67 lay out the `WebToolBar` and `WebBrowserPane` in the content pane. A `JInternalFrame` has zero size when first created, so line 70 invokes method `setSize` of class `JInternalFrame` to size the `JInternalFrame` appropriately. To prevent new `JInternalFrames` from obscuring other `JInternalFrame` s in the `JDesktopPane`, lines 73–74 invoke method ` setLocation` of class `JInternalFrame` to position the new `JInternalFrame` at an offset from the previously created `JInternalFrame`. Line 77 invokes method `add` of class `JDesktopPane` to add the `JInternalFrame` to the display, and line 80 invokes method `setVisible` of class `JInternalFrame` to make the `JInternalFrame` visible.

**Look-and-Feel Observation 2.2**

`JInternalFrames` have no size and are invisible by default. When creating a new `JInternalFrame`, be sure to invoke method `setSize` to size the `JInternalFrame` and `setVisible` (true) to make the `JInternalFrame` visible.

Class `MDIWebBrowser` uses two `Actions`—`NewWindowAction` for creating new Web browser windows and `ExitAction` for exiting the application. Lines 84–101 declare inner class `NewWindowAction`, which extends class `AbstractAction`. Lines 90–93 invoke method `putValue` of interface `Action` to configure `NewWindowAction` properties. Method `actionPerformed` (lines 97–100) invokes method `createNewWindow` to create a new Web browser window each time the user invokes `NewWindowAction`. Class `ExitAction` (lines 104–120) also invokes method `putValue` to configure the `Action` (lines 110–112) and implements method `actionPerformed` (lines 116–119) to exit the application (line 118) when invoked.

### 2.6 Drag and Drop

Drag and drop is a common way to manipulate data in a GUI. Most GUIs emulate real-world desktops, with icons that represent the objects on a virtual desk. Drag and drop enables users to move items around the desktop and to move and copy data among applications using mouse gestures. A gesture is a mouse movement that corresponds to a drag-and-drop operation, such as dragging a file from one folder and dropping the file into another folder.

Two Java APIs enable drag-and-drop data transfer between applications. The data transfer API—package `java.awt.datatransfer`—enables copying and moving data within a single application or among multiple applications. The drag-and-drop API enables Java applications to recognize drag-and-drop gestures and to respond to drag-and-drop operations. A drag-and-drop operation uses the data transfer API to transfer data from the `drag source` to the `drop target`. For example, a user could begin a drag gesture in a file-manager application (the drag source) to drag a file from a folder and drop the file on a Java application (the drop target). The Java application would use the drag-and-drop API to rec-
ognize that a drag-and-drop operation occurred and would use the data transfer API to
retrieve the data transferred through the drag-and-drop operation.

**DnDWebBrowser** (Fig. 2.11) is a Web-browsing application that also allows users to
drop a file onto the **WebBrowserPane** to view the file contents. For example, a user
could drag an HTML file from the host operating system’s file manager and drop the file
on the **WebBrowserPane** to render the HTML. **DnDWebBrowser** uses the drag-and-
drop API to recognize drag-and-drop operations and the data transfer API to retrieve the
transferred data. Lines 32–33 create a **WebBrowserPane** component for viewing Web
pages and a **WebToolBar** to provide navigation controls.

The **WebBrowserPane** in class **DnDWebBrowser** acts as a drop target (i.e., a user
can drop a dragged object on the **WebBrowserPane**). Lines 37–38 invoke method **set-
DropTarget** of class **WebBrowserPane** and create a new **DropTarget** object. The
first argument to the **DropTarget** constructor is the **java.awt.Component** that
provides the GUI target onto which a user can drop objects. In this case, the **Component** is a
**WebBrowserPane**. The second argument specifies the types of drag-and-drop operations
that the **DropTarget** supports. Class **DnDConstants** specifies constant
**ACTION_COPY** for allowing a **DropTarget** to accept a drag-and-drop operation for
copying a dragged object. Other operations include **ACTION_MOVE** for moving an object
and **ACTION_LINK** for creating a link to an object (e.g., a symbolic link on a UNIX file-
system). The third argument to the **DropTarget** constructor is the **DropTargetLis-
tener** to be notified of drag-and-drop operation events.

Class **DropTargetHandler** (lines 48–126) implements interface **DropTar-
getListener** to listen for drag-and-drop operation events related to a **DropTarget**.
The drag-and-drop subsystem invokes method **drop** (lines 51–100) of interface
**DropTargetListener** when the user drops an object on a **DropTarget**. Line 54
invokes method **getTransferable** of class **DropTargetDropEvent** to retrieve the
**Transferable** object that the user dropped. Interface **java.awt.data-
transfer.Transferable** declares methods that represent an object that can be trans-
ferred among applications. As part of the datatransfer API, interface **Transferable** rep-
resents objects that may be transferred through the system clipboard (e.g., via cut-and-paste
operations) and objects that are transferred through drag and drop.

```java
// DnDWebBrowser.java
// DnDWebBrowser is an application for viewing Web pages using
drag and drop.
package com.deitel.advjhtp1.gui.dnd;

// Java core packages
import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.util.*;
import java.io.*;
import java.net.*;
```

**Fig. 2.11** **DnDWebBrowser** application for browsing Web sites that also accepts
drag-and-drop operations for viewing HTML pages (part 1 of 5).
// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

// Deitel packages
import com.deitel.advjhtp1.gui.webbrowser.*;

public class DnDWebBrowser extends JFrame {

    private WebToolBar toolBar;
    private WebBrowserPane browserPane;

    // DnDWebBrowser constructor
    public DnDWebBrowser()
    {
        super( "Drag-and-Drop Web Browser" );

        // create WebBrowserPane and WebToolBar for navigation
        browserPane = new WebBrowserPane();
        toolBar = new WebToolBar( browserPane );

        // enable WebBrowserPane to accept drop operations, using
        // DropTargetHandler as the DropTargetListener
        browserPane.setDropTarget( new DropTarget( browserPane,
                        DnDConstants.ACTION_COPY, new DropTargetHandler() ) );

        // lay out WebBrowser components
        Container contentPane = getContentPane();
        contentPane.add( toolBar, BorderLayout.NORTH );
        contentPane.add( new JScrollPane( browserPane ),
                        BorderLayout.CENTER );
    }

    // inner class to handle DropTargetEvents
    private class DropTargetHandler implements DropTargetListener {

        // handle drop operation
        public void drop( DropTargetDropEvent event )
        {
            // get dropped Transferable object
            Transferable transferable = event.getTransferable();

            // if Transferable is a List of Files, accept drop
            if ( transferable.isDataFlavorSupported( DataFlavor.javaFileListFlavor ) ) {

                // accept the drop operation to copy the object
                event.acceptDrop( DnDConstants.ACTION_COPY );

                // process list of files and display each in browser
                try {
                    // code to process files and display in browser
                }
            }
        }
    }

    // code for processing files and displaying in browser
}

Fig. 2.11  DnDWebBrowser application for browsing Web sites that also accepts drag-and-drop operations for viewing HTML pages (part 2 of 5).
// get List of Files
java.util.List fileList =
    ( java.util.List ) transferable.getTransferData(
        DataFlavor.javaFileListFlavor );

Iterator iterator = fileList.iterator();

while ( iterator.hasNext() ) {
    File file = ( File ) iterator.next();

    // display File in browser and complete drop
    browserPane.goToURL( file.toURL() );
}

// indicate successful drop
event.dropComplete( true );

// handle exception if DataFlavor not supported
catch ( UnsupportedFlavorException flavorException ) {
    flavorException.printStackTrace();
    event.dropComplete( false );
}

// handle exception reading Transferable data
catch ( IOException ioException ) {
    ioException.printStackTrace();
    event.dropComplete( false );
}

// if dropped object is not file list, reject drop
else
    event.rejectDrop();

// handle drag operation entering DropTarget
public void dragEnter( DropTargetDragEvent event ) {
    // if data is javaFileListFlavor, accept drag for copy
    if ( event.isDataFlavorSupported(
        DataFlavor.javaFileListFlavor ) )
        event.acceptDrag( DnDConstants.ACTION_COPY );
    else
        event.rejectDrag();

    // invoke when drag operation exits DropTarget
    public void dragExit( DropTargetEvent event ) {}
// invoked when drag operation occurs over DropTarget
public void dragOver( DropTargetDragEvent event ) {}

// invoked if dropAction changes (e.g., from COPY to LINK)
public void dropActionChanged( DropTargetDragEvent event )
{
}

} // end class DropTargetHandler

// execute application
public static void main( String args[] )
{
    DnDWebBrowser browser = new DnDWebBrowser();
browser.setDefaultCloseOperation( EXIT_ON_CLOSE );
browser.setSize( 640, 480 );
browser.setVisible( true );
}

Fig. 2.11  DnDWebBrowser application for browsing Web sites that also accepts
drag-and-drop operations for viewing HTML pages (part 4 of 5).
Lines 57–58 invoke method `isDataFlavorSupported` of interface `Transferable` to determine the type of data the `Transferable` object contains. The datatransfer API defines class `DataFlavor` to represent types of data contained in a `Transferable` object. Class `DataFlavor` provides several `static` constants that developers can use for comparison to common `DataFlavors`. Lines 57–58 determine if the `Transferable` object supports `DataFlavor.javaFileListFlavor`, which represents a `List` of `Files`. If a user drags one or more `Files` from the host operating system’s file manager, the dropped `Transferable` object will support `DataFlavor.javaFileListFlavor`. If the `Transferable` object supports this `DataFlavor`, line 61 invokes method `acceptDrop` of class `DropTargetDropEvent` to indicate that the drop operation is allowed for this `DropTarget`.

Lines 67–69 retrieve the `List` of `Files` from the `Transferable` object by invoking method `getTransferData` of interface `Transferable`. Lines 73–78 iterate the `List` of `Files`, displaying each by invoking method `goToURL` of class `WebBrowserPane`. Line 80 invokes method `dropComplete` of class `DropTargetDropEvent` with a `true` argument to indicate that the drag-and-drop operation was successful. If the `DataFlavor` was not `DataFlavor.javaFileListFlavor`, line 99 invokes method `rejectDrop` of class `DropTargetDropEvent` to reject the drag-and-drop operation.

The drag-and-drop subsystem invokes method `dragEnter` of interface `DropTargetListener` (lines 103–114) when a drag-and-drop operation enters a `DropTarget` (e.g., the user drags the mouse into the `DropTarget`). Lines 106–107 check the `DataFlavors` that the `Transferable` object supports. If the `Transferable` object supports `DataFlavor.javaFileListFlavor`, line 109 invokes method `acceptDrag` of class `DropTargetDragEvent` to indicate that this
DropTarget allows the drag-and-drop operation. If the Transferable object does not support DataFlavor.javaFileListFlavor, line 113 invokes method rejectDrag of class DropTargetDragEvent to indicate that the DropTarget does not allow this drag-and-drop operation. The operating system may provide a visual cue to the user to indicate that the DropTarget does not allow the drag-and-drop operation, for example, by changing the mouse cursor.

The drag-and-drop subsystem invokes method dragExit (line 117) of interface DropTargetListener when the drag-and-drop operation leaves the DropTarget and method dragOver (line 120) when the drag-and-drop operation passes over the DropTarget. If the user changes the drop action (e.g., from DndConstants.ACTION_COPY to DndConstants.ACTION_MOVE by pressing the Ctrl key), the drag-and-drop subsystem invokes method dropActionChanged (line 123). We provide empty implementations of these methods because we do not require any special handling for these events.

2.7 Internationalization

Internationalization is the process of preparing an application for distribution in multiple locales. A locale identifies the language, currency, character set, date formats and other items most widely used for presenting information in a particular country or region. For example, in the U.S. locale, the language is English, the currency is the U.S. dollar and the date format is month/day/year. In the United Kingdom locale, the language also is English, but the currency is the British pound and the date format is day/month/year. Applications to be distributed in multiple locales must display information in the correct language and with appropriate date, currency and other formats.

To internationalize an application, a developer must replace hard-coded strings that the user might see, such as labels, tooltips and error messages, with strings contained in a ResourceBundle. A ResourceBundle is a Java properties file that maps keys to string values. For example, a ResourceBundle could contain the key exitButtonLabel with the string value Exit. Instead of hard coding the string Exit on a JButton’s label, the developer could retrieve the label from the ResourceBundle. The developer could then provide multiple versions of the ResourceBundle that use the same keys, but provide string values in different languages. For example, the developer could provide a ResourceBundle that contains French translations of each string value.

The developer also must use locale-sensitive classes to format data, such as dates, times and currencies, using locale-specific formats. There are several locale-sensitive classes that can perform this formatting, such as NumberFormat and DateFormat. A locale-sensitive class uses information about the appropriate locale to produce its output. For example, method format of class DateFormat takes as arguments a Date and a Locale and returns an appropriately formatted String for the given Locale (e.g., the string 3/8/2001 for the U.S. Locale).

Internationalized applications also must use Unicode characters. Unicode is a standard for encoding characters for most of the world’s languages. Java uses Unicode to represent all characters, but it is possible that data generated by other applications may not use Unicode. Such data would need to be converted to Unicode before including it in an internationalized application. For more information about Unicode, please see Appendix I, Unicode.
Figure 2.12 presents an internationalized WebToolBar class. The WebToolBar constructor (lines 27–104) takes as an additional argument the Locale for which the WebToolBar should be localized. Lines 30–31 load the ResourceBundle named StringsAndLabels for the given Locale by invoking static method getBundle of class ResourceBundle. Line 33 invokes method getString of class ResourceBundle to retrieve the toolBarTitle string from the ResourceBundle. Line 33 also invokes method setName of class JToolBar to set the JToolBar’s name to the retrieved value.

```java
// WebToolBar.java
// Internationalized WebToolBar with components for navigating
// a WebBrowserPane.
package com.deitel.advjhtp1.gui.i18n;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

// Deitel packages
import com.deitel.advjhtp1.gui.webbrowser.WebBrowserPane;
import com.deitel.advjhtp1.gui.actions.MyAbstractAction;

public class WebToolBar extends JToolBar
   implements HyperlinkListener {
   private WebBrowserPane webBrowserPane;
   private JTextField urlTextField;

   // WebToolBar constructor
   public WebToolBar( WebBrowserPane browser, Locale locale )
   {
      // get resource bundle for internationalized strings
      ResourceBundle resources = ResourceBundle.getBundle("StringsAndLabels", locale);
      setName( resources.getString( "toolBarTitle" ) );

      // register for HyperlinkEvents
      webBrowserPane = browser;
      webBrowserPane.addHyperlinkListener( this );

      // create JTextField for entering URLs
      urlTextField = new JTextField( 25 );
   }
}
```

Fig. 2.12 WebToolBar that uses ResourceBundles for internationalization (part 1 of 3).
urlTextField.addActionListener(
    new ActionListener() {

        // navigate webBrowser to user-entered URL
        public void actionPerformed( ActionEvent event )
        {
            // attempt to load URL in webBrowserPane
            try {
                URL url = new URL( urlTextField.getText() );
                webBrowserPane.goToURL( url );
            }
            // handle invalid URL
            catch ( MalformedURLException urlException ) {
                urlException.printStackTrace();
            }
        }
    });

    // create backAction and configure its properties
    MyAbstractAction backAction = new MyAbstractAction() {

        public void actionPerformed( ActionEvent event )
        {
            // navigate to previous URL
            URL url = webBrowserPane.back();

            // display URL in urlTextField
            urlTextField.setText( url.toString() );
        }
    };
    backAction.setSmallIcon( new ImageIcon( getClass().getResource( "images/back.gif" ) ) );
    backAction.setShortDescription( resources.getString( "backToolTip" ) );

    // create forwardAction and configure its properties
    MyAbstractAction forwardAction = new MyAbstractAction() {

        public void actionPerformed( ActionEvent event )
        {
            // navigate to next URL
            URL url = webBrowserPane.forward();

            // display new URL in urlTextField
            urlTextField.setText( url.toString() );
        }
    };

Fig. 2.12  WebToolBar that uses ResourceBundle for internationalization (part 2 of 3).
Lines 62–78 create an instance of class `MyAbstractAction` (Fig. 2.13) for the `WebToolBar`'s `backAction`. Lines 64–71 implement method `actionPerformed`. Lines 74–75 load the `Icon` for `backAction`, and lines 77–78 retrieve the internationalized tooltip text for `backAction` from the `ResourceBundle`. Lines 81–97 create the `forwardAction` in a similar manner.

The internationalized `WebToolBar` class also replaces the forward and back `JButtons` with `Actions`. Abstract class `MyAbstractAction` (Fig. 2.13) extends class `AbstractAction` to provide `set` methods for commonly used `Action` properties. The `MyAbstractAction` constructor (lines 19–27) takes as arguments the name, `Icon`, description and mnemonic key for the `Action`. Lines 23–26 invoke the appropriate `set` methods to configure the `Action` to the given values. Each `set` method invokes method `putValue` of interface `Action` with the appropriate key and the given value.

Figure 2.14 presents an internationalized `WebBrowser` class. Class `WebBrowser` has a single user-visible string, which is the application window title. The `WebBrowser` constructor (lines 26–42) takes as an argument the `Locale` for which the application should be localized. Lines 28–29 invoke static method `getBundle` of class `Resour-
ceBundle to load the ResourceBundle containing the appropriate internationalized strings. Line 31 invokes method getString of class ResourceBundle to retrieve the applicationTitle string.

```java
// MyAbstractAction.java
// MyAbstractAction is an AbstractAction subclass that provides
// set methods for Action properties (e.g., name, icon, etc.).
package com.deitel.advjhtp1.gui.actions;

// Java core packages
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public abstract class MyAbstractAction extends AbstractAction {

    // no-argument constructor
    public MyAbstractAction() {};

    // construct MyAbstractAction with given name, icon
    // description and mnemonic key
    public MyAbstractAction( String name, Icon icon,
                              String description, Integer mnemonic )
    {
        // initialize properties
        setName( name );
        setSmallIcon( icon );
        setShortDescription( description );
        setMnemonic( mnemonic );
    }

    // set Action name
    public void setName( String name )
    {
        putValue( Action.NAME, name );
    }

    // set Action Icon
    public void setSmallIcon( Icon icon )
    {
        putValue( Action.SMALL_ICON, icon );
    }

    // set Action short description
    public void setShortDescription( String description )
    {
        putValue( Action.SHORT_DESCRIPTION, description );
    }

Fig. 2.13 MyAbstractAction AbstractAction subclass that provides set methods for Action properties (part 1 of 2).
```
// set Action mnemonic key
public void setMnemonic( Integer mnemonic )
{
    putValue( Action.MNEMONIC_KEY, mnemonic );
}

// abstract actionPerformed method to be implemented
// by concrete subclasses
public abstract void actionPerformed( ActionEvent event );

// WebBrowser.java
// WebBrowser is an application for browsing Web sites using
// a WebToolBar and WebBrowserPane.
package com.deitel.advjhtp1.gui.i18n;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

// Deitel packages
import com.deitel.advjhtp1.gui.webbrowser.WebBrowserPane;

public class WebBrowser extends JFrame {
    private ResourceBundle resources;
    private WebToolBar toolBar;
    private WebBrowserPane browserPane;

    // WebBrowser constructor
    public WebBrowser( Locale locale )
    {
        resources = ResourceBundle.getBundle(
            "StringsAndLabels", locale );
        setTitle( resources.getString( "applicationTitle" ) );
        // create WebBrowserPane and WebToolBar for navigation
        browserPane = new WebBrowserPane();
        toolBar = new WebToolBar( browserPane, locale );
    
    // WebBrowser that uses ResourceBundles for internationalization
    (part 1 of 2).

    // MyAbstractAction AbstractAction subclass that provides set
    methods for Action properties (part 2 of 2).
Fig. 2.14 WebBrowser that uses ResourceBundle for internationalization (part 2 of 2).

Class BrowserLauncher (Fig. 2.15) provides a JComboBox for selecting a Locale and launching an internationalized WebBrowser. Line 25 creates the JComboBox and lines 28–34 add sample Locales to the JComboBox. When the user selects a Locale from the JComboBox, lines 43–44 invoke method launchBrowser of class BrowserLauncher to launch a new WebBrowser. Method launchBrowser (lines 57–63) creates a new WebBrowser for the given Locale, sets its size and displays it.

The properties files of Fig. 2.16 and Fig. 2.17 contain internationalized strings for the default Locale (Locale.US) and the French Locale (Locale.FRANCE). In a properties file, the # character begins a single-line comment. Each property has a key, followed by an equals sign, followed by a value.

Note in Fig. 2.17 that the backToolTip value represents special characters (e.g., characters with accents) as Unicode escape sequences (line 3). Unicode can represent over 65,000 unique characters. A Unicode escape sequence begins with \u and contains four hexadecimal digits that represent the special character. Java uses Unicode characters by default and requires Unicode characters for proper internationalization.

Fig. 2.15 BrowserLauncher application for selecting a Locale and launching an internationalized WebBrowser (part 1 of 3).
super( "Browser Launcher" );

// create JComboBox and add Locales
localeComboBox = new JComboBox();

// United States, English
localeComboBox.addItem( Locale.US );

// France, French
localeComboBox.addItem( Locale.FRANCE );

// Russia, Russian
localeComboBox.addItem( new Locale( "ru", "RU" ) );

// launch new WebBrowser when Locale selection changes
localeComboBox.addItemListener(
    new ItemListener() {
        public void itemStateChanged( ItemEvent event ) {
            if ( event.getStateChange() == ItemEvent.SELECTED )
                launchBrowser( ( Locale )
                    localeComboBox.getSelectedItem() );
        }
    });

// lay out components
Container contentPane = getContentPane();
contentPane.setLayout( new FlowLayout() );
contentPane.add( new JLabel( "Select Locale" ) );
contentPane.add( localeComboBox );

// launch new WebBrowser for given Locale
private void launchBrowser( Locale locale ) {
    WebBrowser browser = new WebBrowser( locale );
    browser.setDefaultCloseOperation( DISPOSE_ON_CLOSE );
    browser.setSize( 640, 480 );
    browser.setVisible( true );
}

// execute application
public static void main( String args[] ) {
    BrowserLauncher launcher = new BrowserLauncher();
    launcher.setDefaultCloseOperation( EXIT_ON_CLOSE );
    launcher.setSize( 200, 125 );
    launcher.setVisible( true );
}

Fig. 2.15 BrowserLauncher application for selecting a Locale and launching an internationalized WebBrowser (part 2 of 3).
Fig. 2.15  **BrowserLauncher** application for selecting a **Locale** and launching an internationalized **WebBrowser** (part 3 of 3).

```java
1  # English language strings for internationalized WebBrowser
2  # application title
3  applicationTitle = Deitel Web Browser
4
5  # title for WebToolBar
6  toolBarTitle = Web Navigation
7```

Fig. 2.16  Properties file for default **Locale** (US English)—**StringsAndLabels.properties** (part 1 of 2).
The filenames for properties files enable internationalized applications to load the proper resources for the selected Locale. Note the names in the above figure captions for the properties files. The properties file for the default Locale (i.e., the Locale used if there is none specified) is named StringsAndLabels.properties. The properties file for Locale.FRANCE is named StringsAndLabels_fr_FR.properties. This name specifies that this is an internationalized version of the StringsAndLabels properties file for the French language (fr) in the country of France (FR). The lowercase language abbreviation is an ISO Language Code for the French language. The uppercase country abbreviation is an ISO Country Code for the country of France. Together, the ISO Language Code and ISO Country Code specify a locale. The list of ISO Language codes is available at www.ics.uci.edu/pub/ietf/http/related/iso639.txt. The list of ISO Country Codes is available at www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.

### 2.8 Accessibility

Accessibility refers to the level of an application’s usability for people with disabilities. To make an application accessible means to ensure that the application works for people with disabilities. Many software applications are inaccessible to people with visual, learning or mobility impairments. A high level of accessibility is difficult to achieve because there are many different disabilities, language barriers, hardware and software inconsistencies and so on. As greater numbers of people use computers, it is imperative that application designers increase the accessibility of their applications. Recent legislation in the United States has brought accessibility to the forefront of Web and application development.
The Swing API designers took advantage of the Java Accessibility API to build accessibility features into every Swing component to facilitate creating accessible Java applications. As a result, Java developers who use the Swing APIs to build application GUIs need only use the Swing APIs properly to enable most accessibility features. For example, when creating GUI elements such as JButton and JMenuItem, developers should provide tooltip text that describes the component and mnemonic keys or accelerator keys for enabling keyboard access. These simple properties enable accessibility tools, such as screen readers, to convey important descriptive information to the user. Enabling keyboard access makes applications easier to navigate for all users, and also allows accessibility tools to navigate the application more easily.

When it is not appropriate for a GUI component to have a tooltip or label, developers can use methods setAccessibleName and setAccessibleDescription of class AccessibilityContext to provide descriptive text. Each Swing component contains an AccessibilityContext for enabling the component’s accessibility features. Assistive technologies (e.g., screen readers, input devices) then use the Java Access Bridge to interact with the Java application to take advantage of the developer-provided descriptive text.

Class ActionSample2 (Fig. 2.18) modifies class ActionSample (Fig. 2.5) to demonstrate adding accessible component names and descriptions to Swing components. Action actionSample (lines 26–50) now contains accessible text in the dialog box that opens when actionSample is fired. Lines 36–37 declare an AccessibilityContext object for the JOptionPane action by calling method getAccessibleContext on action. Line 38 calls method setAccessibleName to set action’s name in AccessibleContext actionContext. Lines 39–41 call method setAccessibleDescription of class AccessibilityContext to set actionSample’s description. Line 53 specify a name for actionSample and lines 60–61 specify a short description. Lines 64–65 assign a mnemonic key to actionSample.

Action exitAction (lines 68–92) now contains accessible text in the dialog box that opens when exitSample is fired. Lines 78–79 obtain an AccessibilityContext for the JOptionPane by invoking method getAccessibleContext. Line 80 calls method setAccessibleName to specify a name for the JOptionPane’s AccessibleContext. Lines 81–83 call method setAccessibleContext. Line 96 specifies a name for exitAction by invoking method putValue of interface Action. Lines 102–103 associate a short description with exitAction. Lines 106–107 assign a mnemonic key to exitAction.

Fig. 2.18 ActionSample2 demonstrates Accessibility package (part 1 of 5).
Chapter 2  Advanced Swing Graphical User Interface Components  73

```
10 // Java extension packages
11 import javax.accessibility.*;
12 import javax.swing.*;
13
14 public class ActionSample2 extends JFrame {
15
16   // Swing Actions
17   private Action sampleAction;
18   private Action exitAction;
19
20   // ActionSample2 constructor
21   public ActionSample2()
22   {
23      super( "Using Actions" );
24
25      // create AbstractAction subclass for sampleAction
26      sampleAction = new AbstractAction() {
27         public void actionPerformed( ActionEvent event )
28         {
29            // display message indicating sampleAction invoked
30            JOptionPane action = new JOptionPane(
31                 "The sampleAction was invoked." );
32
33            // get AccessibleContext for action and set name
34            // and description
35            AccessibleContext actionContext =
36               action.getAccessibleContext();
37            actionContext.setAccessibleName( "sampleAction" );
38            actionContext.setAccessibleDescription(
39                  "SampleAction opens a dialog box to demonstrate"
40                  + " the Action class." );
41
42            // create and display dialog box
43            action.createDialog( ActionSample2.this,
44                 "sampleAction" ).setVisible( true );
45
46            // enable exitAction and associated GUI components
47            exitAction.setEnabled( true );
48         }
49      }
50   };
51
52   // set Action name
53   sampleAction.putValue( Action.NAME, "Sample Action" );
54
55   // set Action Icon
56   sampleAction.putValue( Action.SMALL_ICON, new ImageIcon(
57                  getClass().getResource( "images/Help24.gif" ) ) );
58
59   // set Action short description (tooltip text)
60   sampleAction.putValue( Action.SHORT_DESCRIPTION,
61                  "A Sample Action" );
62
Fig. 2.18  ActionSample2 demonstrates Accessibility package (part 2 of 5).
```
// set Action mnemonic key
sampleAction.putValue( Action.MNEMONIC_KEY,
    new Integer( 'S' ) );

// create AbstractAction subclass for exitAction
exitAction = new AbstractAction() {

    public void actionPerformed( ActionEvent event )
    {
        // display message indicating sampleAction invoked
        JOptionPane exit = new JOptionPane(
            "The exitAction was invoked." );

        // get AccessibleContext for exit and set name and
        // description
        AccessibleContext exitContext =
            exit.getAccessibleContext();
        exitContext.setAccessibleName( "exitAction" );
        exitContext.setAccessibleDescription( "ExitAction" + " opens a dialog box to demonstrate the" + " Action class and then exits the program." );

        // create and display dialog box
        exit.createDialog( ActionSample2.this,
            "exitAction" ).setVisible( true );

        // exit program
        System.exit( 0 );
    }
};

// set Action name
exitAction.putValue( Action.NAME, "Exit" );

// set Action icon
exitAction.putValue( Action.SMALL_ICON, new ImageIcon( getClass().getResource( "images/EXIT.gif" ) ) );

// set Action short description (tooltip text)
exitAction.putValue( Action.SHORT_DESCRIPTION,
    "Exit Application" );

// set Action mnemonic key
exitAction.putValue( Action.MNEMONIC_KEY,
    new Integer( 'x' ) );

// disable exitAction and associated GUI components
exitAction.setEnabled( false );

// create File menu
JMenu fileMenu = new JMenu( "File" );

---

Fig. 2.18 ActionSample2 demonstrates Accessibility package (part 3 of 5).
// add sampleAction and exitAction to File menu to
// create a JMenuItem for each Action
fileMenu.add( sampleAction );
fileMenu.add( exitAction );

fileMenu.setMnemonic( 'F' );

// create JMenuBar and add File menu
JMenuBar menuBar = new JMenuBar();
menuBar.add( fileMenu );
setJMenuBar( menuBar );

// create JToolBar
JToolBar toolBar = new JToolBar();

// add sampleAction and exitAction to JToolBar to create
// JButtons for each Action
toolBar.add( sampleAction );
toolBar.add( exitAction );

// get AccessibleContext for toolBar and set name and
daemons
AccessibleContext toolContext =
toolBar.getAccessibleContext();
toolContext.setAccessibleName( "ToolBar" );
toolContext.setAccessibleDescription( "ToolBar contains
 + " sampleAction button and exitAction button." );

// create JButton and set its Action to sampleAction
JButton sampleButton = new JButton();
sampleButton.setAction( sampleAction );

// get AccessibleContext for sampleButton and set name
// and description
AccessibleContext sampleContext =
sampleButton.getAccessibleContext();
sampleContext.setAccessibleName( "SampleButton" );
sampleContext.setAccessibleDescription( "SampleButton
 + " produces a sampleAction event." );

// create JButton and set its Action to exitAction
JButton exitButton = new JButton( exitAction );

// get AccessibleContext for exitButton and set name and
// description
AccessibleContext exitContext =
exitButton.getAccessibleContext();
exitContext.setAccessibleName( "ExitButton" );
exitContext.setAccessibleDescription( "ExitButton
 + " produces an exitAction event." );

Fig. 2.18 ActionSample2 demonstrates Accessibility package (part 4 of 5).

We will download the Java Access Bridge and a demonstration of JAWS for Windows 3.7 to demonstrate the accessibility features of ActionSample2. The Java Access Bridge allows assistive programs in Windows to use the accessibility information of a Java program. The Java Access Bridge can be downloaded at

java.sun.com/products/accessbridge

JAWS for Windows is a screen reader from Henter-Joyce (www.hj.com). A demonstration version of JAWS can be downloaded at


Download and install both programs to try the rest of the example in this section.

With the Access Bridge installed and JAWS running in the background, execute ActionSample2. JAWS reads the name of the new window that opens. The GUI of ActionSample2 (Fig. 2.19) is identical to the original ActionSample (Fig. 2.5).
Switch between the buttons by pressing Tab to move forward or Shift + Tab to move backward. JAWS reads the name of the new button whenever the focus changes. To press the button that holds the focus, press the space bar. The sampleAction dialog opens and JAWS reads its name. Pressing the space bar or the Enter key closes the dialog. The Exit button is now available in the GUI. Switch the focus to the larger button labeled Sample Action (not the one in the tool bar) and press Insert + F1. This JAWS command reads the description attached to the button’s AccessibleContext (Fig. 2.20). Do the same command on the Exit button to hear its description (Fig. 2.21). ActionSample2’s Actions are also available through the File menu. The File menu’s mnemonic key is the underlined letter F. Pressing Alt + F opens the File menu and causes JAWS to read the menu name (Fig. 2.22). The arrow keys move the cursor within the menu. JAWS reads the name of each menu item as it is selected (Fig. 2.23). Pressing the space bar, Enter key, or a mnemonic key activates one of the Actions.

Fig. 2.19 Actions sampleAction and exitAction of ActionSample2.

"SampleButton produces a sampleAction event."

Fig. 2.20 AccessibleDescription of sampleButton.
2.9 Internet and World Wide Web Resources

**Swing**

The Swing Connection contains technical articles and documentation for Swing components.

JavaWorld collection of Swing-related articles.
Internationalization

www.ibm.com/developerworks/theme/international-index.html
IBM offers links to internationalization resources including multilingual software and international calendars.

dev-developer.java.sun.com/developer/technicalArticles/Intl/index.html
This site provides numerous articles on learning how to localize and internationalize various Java programs.

www.onjava.com/pub/a/onjava/2001/04/12/internationalization.html
This article, *Java Internationalization and Localization*, by Jaric Sng describes the steps for accessing, installing, and determining fonts, focusing on Japanese, Chinese, and Korean.

java.sun.com/j2se/1.3/docs/guide/intl
This site supplies a guide to Java internationalization. It includes a detailed section on formatting currencies, time zones, dates, texts, messages and other international dissimilarities.

Accessibility

java.sun.com/products/jfc/jaccess-1.3/doc/guide.html
Sun Microsystems has improved Java accessibility through Java Accessibility API and Java Accessibility Utilities. Check out a detailed description of these packages at this site.

dev-developer.java.sun.com/developer/earlyAccess/jaccesshelper
The Java Accessibility Helper examines Java programs for accessibility issues and provides a report that details changes needed to be made. This is an early-access download and requires a free registration with the Java Developer Connection Web site.

www.ibm.com/able/snsjavag.html
"IBM Guidelines for Writing Accessible Applications Using 100% Pure Java," by Richard S. Schwerdtfeger, states the necessary features that should be provided to create full accessibility. In addition, this online guidebook discusses the various programs to achieve accessibility.

www.sun.com/access/developers/access.quick.ref.html
This site simply emphasizes the importance of accessibility and gives tips on making applications accessible.

www.w3.org/WAI
The World Wide Web Consortium’s Web Accessibility Initiative (WAI) site promotes design of universally accessible Web sites. This site will help you keep up-to-date with current guidelines and forthcoming recommendations for Web accessibility.

www.sun.com/access/gnome
GNOME Developer’s Site provides information on various assistive technologies, such as screen magnifiers and screen keyboards for Linux and Unix platforms that use the GNOME user interface.

www.voice-assistant.com
The Voice Mate V4 assists the blind with using a computer. It speaks the menu options and characters as they are typed.

www.magnifiers.org
On this site, you can find information and downloads for screen magnifiers.

www.voicerecognition.com
This site contains information on various voice-recognition products.

trace.wisc.edu/world/web
This site explains how to make Web sites more accessible to disabled users. It also gives multiple references to other sites on Web accessibility.
www.access-board.gov/508.htm

Electronic version of Section 508 of the Rehabilitation Act, which mandates that government agencies provide accessible electronic access to information from federal agencies.

SUMMARY

• Swing provides three basic types of text components for presenting and editing text. Class `JTextComponent` is the base class for all Swing text components, including `JTextField`, `JTextArea` and `JEditorPane`.

• `JTextField` is a single-line text component suitable for obtaining simple user input or displaying information such as form field values, calculation results and so on. `JPasswordField` is a subclass of `JTextField` suitable for obtaining user passwords.

• `JEditorPane` provides enhanced text-rendering capabilities. `JEditorPane` supports styled documents that include formatting, font and color information. `JEditorPane` is capable of rendering HTML documents as well as Rich Text Format (RTF) documents.

• Toolbars are GUI containers typically located below an application’s menus. Toolbars contain buttons and other GUI components for commonly used features, such as cut, copy and paste, or navigation buttons for a Web browser.

• Class `javax.swing.JToolBar` enables developers to add toolbars to Swing user interfaces. `JToolBar` also enables users to modify the appearance of the `JToolBar` in a running application.

• Users can drag a `JToolBar` from the top of a window and ”dock” the `JToolBar`s on the side or bottom. Users also can drag the `JToolBar` away from the application window to create a floating `JToolBar`.

• Based on `JToolBar`’s inheritance hierarchy, each `JToolBar` also is a `java.awt.Container` and therefore can contain other GUI components.

• A `JToolBar` has property `orientation` that specifies how the `JToolBar` will arrange its child components. The default is horizontal orientation, which indicates that the `JToolBar` lays out its child components next to one another.

• The Command design pattern enables developers to define requests (e.g., a user request to copy text) once in a reusable object that the developer then can add to a menu, toolbar or pop-up menu. This design pattern is called Command because it defines a user command or instruction.

• An `Action`, which implements the Command design pattern, represents user-interface logic and properties for GUI components that represent that logic, such as the label for a button, the text for a tool tip and the mnemonic key for keyboard access.

• The logic for an `Action` takes the form of an `actionPerformed` method that the event mechanism invokes in response to the user activating an interface component (e.g., the user clicking a `JButton`).

• Interface `Action` extends interface `ActionListener`, which enables `Actions` to process `ActionEvents` generated by GUI components. `Actions` provide an additional benefit in that the developer can enable or disable all GUI components associated with an `Action` by enabling or disabling the `Action` itself.

• that `sampleAction` was invoked. Line 33 then invokes method `setEnabled` of interface `Action` on the `exitAction` reference. This enables the `exitAction` and its associated GUI components. Note that `Actions` are enabled by default. We disabled the `exitAction` (line 80) to demonstrate that this disables the GUI components associated with that `Action`.

• `JSplitPane` and `JTabbedPane` are container components that enable developers to present large amounts of information in a small screen area.
• **JSplitPane** divides two components with a divider that users can reposition to expand and contract the visible areas of the **JSplitPane**’s child components. **JSplitPanes** can contain only two child components, although each child component may contain nested components.

• The constant **JSplitPane.HORIZONTAL_SPLIT** specifies the **JSplitPane** should display its child components side-by-side. The constant **JSplitPane.VERTICAL_SPLIT** specifies that the **JSplitPane** should display its child components one on top of the other.

• Adding child components to **JScrollPane**s before adding those components to a **JSplitPane** ensures that the user will be able to view all the content in each child component by scrolling if necessary.

• **JTabbedPane** presents multiple components in separate tabs, which the user can navigate using a mouse or keyboard. Dialog boxes often use components similar to **JTabbedPane**s.

• Multiple document interfaces allow users to view multiple documents in a single application. Each document appears in a separate window in the application. The user can arrange, resize, iconify (i.e., minimize) and maximize these separate document windows like application windows on the desktop.

• **JInternalFrames** behave much like **JFrames**. Users can maximize, iconify, resize, open and close **JInternalFrames**. **JInternalFrames** have title bars with buttons for iconifying, maximizing and closing. Users also can move **JInternalFrames** within the **JDesktopPane**.

• **JInternalFrames** have no size and are invisible by default. When creating a new **JInternalFrame**, be sure to invoke method **setSize** to size the **JInternalFrame** and **setVisible(true)** to make the **JInternalFrame** visible.

• Drag and drop enables users to move items around the desktop and to move and copy data among applications using mouse gestures. A gesture is a mouse movement that corresponds to a drag-and-drop operation, such as dragging a file from one folder and dropping the file into another folder.

• The data transfer API—package **java.awt.datatransfer**—enables copying and moving data within a single application or among multiple applications. The drag-and-drop API enables Java applications to recognize drag-and-drop gestures and to respond to drag-and-drop operations.

• A drag-and-drop operation uses the data transfer API to transfer data from the drag source to the drop target. Applications can use the drag-and-drop API to recognize drag-and-drop operations and use the data transfer API to retrieve the data transferred through those drag-and-drop operations.

• The drag-and-drop subsystem invokes method **drop** of interface **DropTargetListener** when the user drops an object on a **DropTarget**.

• Interface **java.awt.datatransfer.Transferable** declares methods that represent an object that can be transferred among applications. As part of the datatransfer API, interface **Transferable** represents objects that may be transferred through the system clipboard (e.g., via cut-and-paste operations) and objects that are transferred through drag and drop.

• Internationalization is the process of preparing an application for distribution in multiple locales. A locale identifies the language, currency, character set, date formats and other items most widely used for presenting information in a particular country or region.

• Applications to be distributed in multiple locales must display information in the correct language and with appropriate date, currency and other formats.

• A **ResourceBundle** is a Java properties file that maps keys to string values. For example, a **ResourceBundle** could contain the key **exitButtonLabel** with the string value **Exit**. Instead of hard coding the string **Exit** on a **JButton**’s label, the developer could retrieve the label from the **ResourceBundle**. The developer could then provide multiple versions of the **ResourceBundle** that use the same keys, but provide string values in different languages.
• The developer also must use locale-sensitive classes to format data, such as dates, times and currencies, using locale-specific formats. There are several locale-sensitive classes that can perform this formatting, such as {	exttt{NumberFormat}} and {	exttt{DateFormat}}.

• Internationalized applications also must use Unicode characters. Unicode is a standard for encoding characters for most of the world’s languages. Java uses Unicode to represent all characters.

• The filenames for properties files enable internationalized applications to load the proper resources for the selected {	exttt{Locale}}. These filenames must use a lowercase language abbreviation—called an ISO Language Code—and an uppercase country abbreviation—called an ISO Country Code.

• {	extit{Accessibility}} refers to the level of an application’s usability for people with disabilities. To make an application accessible means to ensure that the application works for people with disabilities.

• Many software applications are inaccessible to people with visual, learning or mobility impairments. A high level of accessibility is difficult to achieve because there are many different disabilities, language barriers, hardware and software inconsistencies and so on.

• Recent legislation in the United States has brought accessibility to the forefront of Web and application development.

• The Swing API designers took advantage of the Java Accessibility API to build accessibility features into every Swing component to facilitate creating accessible Java applications. As a result, Java developers who use the Swing APIs to build application GUIs need only use the Swing APIs properly to enable most accessibility features.

• Developers should provide tooltip text that describes each component and mnemonic keys or accelerator keys for enabling keyboard access. These simple properties enable accessibility tools, such as screen readers, to convey important descriptive information to the user. Enabling keyboard access makes applications easier to navigate for all users, and also allows accessibility tools to navigate the application more easily.

• Methods {	exttt{setAccessibleName}} and {	exttt{setAccessibleDescription}} of class {	exttt{AccessibilityContext}} enable developers to provide descriptive text for components. Each Swing component contains an {	exttt{AccessibilityContext}} for enabling the component’s accessibility features.

• Assistive technologies (e.g., screen readers, input devices) can use the Java Access Bridge to interact with Java applications to take advantage of the developer-provided descriptive text.

**TERMINOLOGY**

- {	exttt{AbstractAction}} class
- {	exttt{AccessibleContext}} class
- {	exttt{Action}} interface
- {	exttt{Action.ACCELERATOR_KEY}} constant
- {	exttt{Action.ACTION_COMMAND_KEY}} constant
- {	exttt{Action.MNEMONIC_KEY}} constant
- {	exttt{Action.NAME}} constant
- {	exttt{Action.SHORT_DESCRIPTION}} constant
- {	exttt{Action.SMALL_ICON}} constant
- {	exttt{addHyperlinkListener}} method of class {	exttt{JEditorPane}}
- {	exttt{Command design pattern}}
- {	exttt{DataFlavor.javaFileListFlavor}} constant
- {	exttt{DnDConstants}} class
- {	exttt{drag and drop}}
- {	exttt{drag-and-drop}}
- {	exttt{drag and drop gesture}}
- {	exttt{drag-and-drop}}
- {	exttt{drag-and-drop operation}}
- {	exttt{DropTarget}} class
- {	exttt{DropTargetDragEvent}} class
- {	exttt{DropTargetDropEvent}} class
- {	exttt{DropTargetListener}} interface
- {	exttt{HyperlinkEvent}} class
- {	exttt{HyperlinkEvent.EventType.ACTIVATED}} constant
- {	exttt{HyperlinkEvent}} interface
- {	exttt{iconify}}
- {	exttt{internationalization}}
- {	exttt{Java Access Bridge}}
- {	exttt{JDesktopPane}} class
- {	exttt{JEditorPane}} class
Chapter 2 Advanced Swing Graphical User Interface Components

SELF-REVIEW EXERCISES

2.1 State which of the following are true and which are false. If false, explain why.

a) The Abstract Windowing Toolkit provides a richer set of components than the Swing component set.
   a) False. Swing provides a richer set of components than the older AWT.

b) Swing provides a pluggable look and feel that enables components to change their appearance.
   b) True.

c) JEditorPane is capable of rendering only plain text, not richly styled text.
   c) False. JEditorPane can render HTML and RTF documents, which can contain rich styling information.

d) Toolbars—implemented by class JToolBar—enable developers to provide users with quick access to commonly used user-interface elements, such as cut, copy and paste.
   d) True.

e) Interface Action provides set and get methods for each Action property.
   e) False. Interface Action provides method putValue, which enables programmers to specify the property name and value as a key/value pair.

f) JSplitPanes can contain any number of child components.
   f) False. Each JSplitPane may contain exactly two child components, but each child component may contain its own child components.

2.2 Fill in the blanks in each of the following:

a) The drag-and-drop API uses the ________ API to transfer data through drag-and-drop operations.
   a) data transfer API.

b) A multiple document interface uses instances of class ________ for individual windows, which are contained in a ________.
   b) JInternalFrame, JDesktopPane.

c) The JInternalFrame constructor takes four boolean arguments that indicate whether the window is ________, ________, ________, and ________.
   c) resizable, closable, max-readable, iconifiable.

d) A ________ identifies the language, currency, character set, date formats and other items most widely used for presenting information in a particular country or region.
   d) Locale.

e) ________ refers to the level of an application’s usability for people with disabilities.
   e) Accessibility.

ANSWERS TO SELF-REVIEW EXERCISES

2.1 a) False. Swing provides a richer set of components than the older AWT. b) True. c) False. JEditorPane can render HTML and RTF documents, which can contain rich styling information. d) True. e) False. Interface Action provides method putValue, which enables programmers to specify the property name and value as a key/value pair. f) False. Each JSplitPane may contain exactly two child components, but each child component may contain its own child components.

2.2 a) data transfer API. b) JInternalFrame, JDesktop Pane. c) resizable, closable, maximize, iconifiable. d) Locale. e) Accessibility.

EXERCISES

2.3 Modify class WebToolBar (Fig. 2.3) to include a JComboBox from which the user can select URLs from the history.
2.4 Create an image-viewing application that supports drag-and-drop loading of images. When the user drags and drops an image file onto the application window, load that image in an ImageIcon and display the ImageIcon in a JPanel.

2.5 Modify class ActionSample2 (Fig. 2.18) to use ResourceBundles for all user-visible Strings in the application. If you know a language other than English, provide a ResourceBundle that contains Strings in that language.

2.6 Making an application accessible requires that the application provides keyboard navigation for all the application’s functionality. Unplug your mouse from your computer and try using various programs, such as word processors, Web browsers and the Java programs in this chapter. What about these applications makes it difficult to navigate without a mouse? Is there functionality that you cannot access using a keyboard?
Model-View-Controller

Objectives

• To understand the model-view-controller (MVC) architecture for separating data, presentation and user input logic.
• To understand the Observer design pattern.
• To understand MVC’s use in Java’s Swing GUI components.
• To understand the default model implementations for Swing components.
• To understand the use of TableModels to represent tabular data for JTables.
• To understand tree data structures and their use as TreeModels for JTrees.

The universe is wider than our views of it.
Henry David Thoreau

Let all your views in life be directed to a solid, however moderate, independence; ...
Junius

I think that I shall never see
A poem as lovely as a tree.
Joyce Kilmer
3.1 Introduction

In this chapter, we introduce the **model-view-controller architecture** (MVC) and its particular application in Java’s Swing classes. The MVC architecture uses object-oriented design principles to modularize applications into data components, presentation components and input-processing components. Data components maintain the raw application data, such as the text of a document in a word processor or the locations of the pieces in a game of chess. The presentation components most commonly provide a visual representation of application data—for example a 3D graphic showing the chessboard and the arrangement of pieces. The input-processing components handle input from the user, such as dragging the mouse to move a piece on the chess board.

MVC has many uses in desktop applications, enterprise applications, simulations and other types of programs. In this chapter, we discuss MVC in general and its variant, the **delegate-model architecture**. We also introduce the Observer design pattern, which is a design pattern built into the MVC architecture. After reading this chapter, you will be able to design your own programs using MVC. You also will be able to take advantage of advanced Swing components that use the delegate-model architecture, such as **JList**, **JTable** and **JTree**.

3.2 Model-View-Controller Architecture

The **model-view-controller architecture** (MVC) separates application data (contained in the **model**) from graphical presentation components (the **view**) and input-processing logic (the **controller**). MVC originally appeared in Smalltalk-80 as a method for separating user interfaces from underlying application data.\(^1\) Figure 3.1 shows the relationships between components in MVC. In our Enterprise Java case study (Chapters 17–20), we will show that MVC is applicable across a wide range of problems and can make applications easier to maintain and extend.

The controller implements logic for processing user input. The model contains application data, and the view generates a presentation of the data stored in the model.

---

user provides some input (e.g., by typing text in a word processor.) the controller modifies the model with the given input. It is important to note that the model contains only the raw application data. In a simple text editor, the model might contain only the characters that make up the document. When the model changes, it notifies the view of the change, so that the view can update its presentation with the changed data. The view in a word processor might display the characters on the screen in a particular font, with a particular size, etc.

MVC does not restrict an application to a single view and controller. In a word processor, for example, there might be two views of a single document model. One view might display the document as an outline, and the other might display the document in a print-preview window. The word processor also may implement multiple controllers, such as a controller for handling keyboard input and a controller for handling mouse selections. If either controller makes a change in the model, both the outline view and the print-preview window show the change immediately, because the model notifies all views of any changes. A developer can provide additional views and controllers for the model without changing the existing components.

Java’s Swing components implement a variation of MVC that combines the view and controller into a single object, called a delegate (Fig. 3.2). The delegate provides both a graphical presentation of the model and an interface for modifying the model. For example, every JButton has an associated ButtonModel for which the JButton is a delegate. The ButtonModel maintains state information, such as whether the JButton is pressed and whether the JButton is enabled, as well as a list of ActionListeners. The JButton provides a graphical presentation (e.g., a rectangle on the screen with a label and a border) and modifies the ButtonModel’s state (e.g., when the user presses the JButton). We discuss several Swing components that implement the delegate-model architecture throughout this chapter.
3.3 Observable Class and Observer Interface

The Observer design pattern enables loose coupling between an object and its dependent objects.\(^2\) Loosely coupled objects interact by invoking methods declared in well-known interfaces, instead of invoking methods declared in particular classes. Using interface methods prevents each object from relying on the other objects’ concrete class type. For example, Java’s event-handling mechanism uses loose coupling to notify objects of events. If an object needs to handle certain events, it implements the appropriate listener interface (e.g., `ActionListener`). Objects that generate events invoke listener interface methods to notify listening objects of events. This loose coupling enables a `JButton`, for example, to send an `ActionEvent` to a `JFrame` subclass that implements `ActionListener`. The `JButton` interacts with the `JFrame` subclass only through method `actionPerformed` of interface `ActionListener`, and not through any method that is specific to the `JFrame` subclass. The `JButton` can send `ActionEvents` to other objects that also implement interface `ActionListener` (e.g., a programmer-defined class or an inner class).

Class `java.util.Observable` represents a model in MVC, or the subject in the Observer design pattern. Class `Observable` provides method `addObserver`, which takes a `java.util.Observer` argument. Interface `Observer` represents the view in MVC and enables loose coupling between an `Observable` object and its `Observers`. When the `Observable` object changes, it notifies each registered `Observer` of the change. The `Observer` can be an instance of any class that implements interface `Observer`; because the `Observable` object invokes methods defined in interface `Observer`, the objects remain loosely coupled. We discuss the details of this interaction in the example that follows.

The example in Fig. 3.4–Fig. 3.10. uses the MVC architecture, class `Observable` and interface `Observer` to implement an `AccountManager` application for managing bank account information. Figure 3.3 illustrates the application’s MVC architecture. The `AccountController` accepts user input in the form of dollar amounts entered in a `JTextField`. The user then selects a `JButton`, either to withdraw or deposit the given amount, and the `AccountController` modifies the `Account` to execute the transaction. Class `Account` is an `Observable` object that acts as the application’s model. When the `AccountController` performs the withdrawal or deposit, the `Account` notifies each view (`AccountTextView`, `AccountBarGraphView` and `AccountPieChartView`) that the `Account` information has changed. Each view updates its display with the modified `Account` information.

Class `Account` (Fig. 3.4) represents a bank account in the `AccountManager` application (Fig. 3.10). Class `Account` extends class `Observable` (line 9) and acts as a model in the application. Class `Account` has `balance` and `name` properties that represent the amount of money in the `Account` and a short description of the `Account`. The `Account` constructor (lines 18–22) initializes the `name` and `balance` properties.

Method `setBalance` (lines 25–35) changes the model by updating the account `balance`. The MVC architecture requires the model to notify its views when the model changes. Line 31 invokes method `setChanged` of class `Observable` to set the model’s `changed` flag. Line 34 invokes method `notifyObservers` of class `Observable` to notify all `Account Observers` (i.e., views) of the change. An `Observable` object

\(^2\) E. Gamma et al., *Design Patterns* (New York: Addison-Wesley Publishing Company, 1995), 293.
must invoke method `setChanged` before invoking method `notifyObservers`. Method `notifyObservers` invokes method `update` of interface `Observer` for each registered `Observer`. Method `getBalance` (lines 38–41) simply returns the current `Account` balance. Method `getBalance` does not modify the model, so method `getBalance` does not invoke `setChanged` or `notifyObservers`.

**Common Programming Error 3.1**

Failing to invoke method `setChanged` before invoking method `notifyObservers` is a logic error. If method `setChanged` has not been invoked, method `notifyObservers` considers the `Observable` object unchanged and will not invoke each `Observer`’s `update` method.

---

**Fig. 3.3**  
AccountManager application MVC architecture.

```java
1 // Account.java
2 // Account is an Observable class that represents a bank
3 // account in which funds may be deposited or withdrawn.
4 package com.deitel.advjhtp1.mvc.account;
5
6 // Java core packages
7 import java.util.Observable;
8
9 public class Account extends Observable {  
10     // Account balance
11     private double balance;
12
13     // readonly Account name
14     private String name;
15
16     // Account constructor
17     public Account( String accountName, double openingDeposit )
18     {
```

---

**Fig. 3.4**  
Account Observable class that represents a bank account (part 1 of 2).
name = accountName;
    setBalance( openingDeposit );
}

// set Account balance and notify observers of change
private void setBalance( double accountBalance )
{
    balance = accountBalance;
    // must call setChanged before notifyObservers to
    // indicate model has changed
    setChanged();
    // notify Observers that model has changed
    notifyObservers();
}

// get Account balance
public double getBalance()
{
    return balance;
}

// withdraw funds from Account
public void withdraw( double amount )
throws IllegalArgumentException
{
    if ( amount < 0 )
        throw new IllegalArgumentException(
            "Cannot withdraw negative amount" );
    // update Account balance
    setBalance( getBalance() - amount );
}

// deposit funds in account
public void deposit( double amount )
throws IllegalArgumentException
{
    if ( amount < 0 )
        throw new IllegalArgumentException(
            "Cannot deposit negative amount" );
    // update Account balance
    setBalance( getBalance() + amount );
}

// get Account name (readonly)
public String getName()
{
    return name;
}
Software Engineering Observation 3.1

Method `notifyObservers` does not guarantee the order in which it notifies `Observer`s. Method `notifyObservers` as implemented in class `Observable` notifies `Observer`s in the order the `Observer`s were registered, but this behavior may be different in `Observable` subclasses or in different Java implementations.

Software Engineering Observation 3.2

Method `notifyObservers` has no relation to methods `notify` and `notifyAll` of class `Object`. Multithreaded programs use methods `notify` and `notifyAll` to wake up Threads waiting to obtain an `Object`'s monitor.

Method `withdraw` (lines 44–53) subtracts the given `amount` from the `Account` balance. If the given `amount` is negative, lines 48–49 throw an `IllegalArgumentException`. Line 52 subtracts the withdrawn `amount` from the current `balance` and invokes method `setBalance` to update the `Account`. Method `setBalance` will notify `Observer`s that the model was changed, so that the `Observer`s can update their displays.

Method `deposit` (lines 56–65) adds the `amount` input to the `Account` balance. If the `amount` is negative, lines 60–61 throw an `IllegalArgumentException`. Line 64 adds the deposit `amount` to the current `balance` and invokes method `setBalance` to update the `Account`. Method `getName` (lines 68–71) returns the `Account` name.

Application `AccountManager` presents `Account` information to the user through three views: `AccountTextView`, `AccountBarGraphView` and `AccountPieChartView`. Each view presents a different visual representation of the `Account` information. `AbstractAccountView` (Fig. 3.5) is an abstract base class for these `Account` views that provides common functionality, such as registering as an `Account` observer. Class `AbstractAccountView` implements interface `Observer`, which allows each `AbstractAccountView` subclass to register as an `Observer` of an `Account`.

```java
// AbstractAccountView.java
// AbstractAccountView is an abstract class that represents
// a view of an Account.
package com.deitel.advjhtp1.mvc.account;

// Java core packages
import java.util.*;
import java.awt.*;

// Java extension packages
import javax.swing.JPanel;
import javax.swing.border.*;

public abstract class AbstractAccountView extends JPanel
    implements Observer {
    // Account to observe
    private Account account;
}
```

Fig. 3.5 `AbstractAccountView` abstract base class for observing `Accounts` (part 1 of 2).
Class **AbstractAccountView** extends **JPanel** because **AbstractAccountView** implementations provide graphical presentations of **Account** data. Line 18 declares a **private** member variable for the **Account** that the **AbstractAccountView** will observe. The constructor (lines 21–37) sets the **account** member variable to the new **Account** (line 29). Line 32 invokes method **addObserver** of class **Observable** to register the newly created **AbstractAccountView** instance as an **Observer** of the new **Account**. The **Account** will now notify this **AbstractAccountView** of any modifications to the **Account**. Lines 35–36 set the **AbstractAccountView**’s background color and border.

Method **getAccount** (lines 40–43) returns the **AbstractAccountView**’s **account**. Method **updateDisplay** (line 46) is marked **abstract**, requiring each **AbstractAccountView** subclass to provide an appropriate implementation for displaying the **Account** information. For example, **AbstractAccountView** subclass **AccountTextView** provides an **updateDisplay** implementation that shows the
Account balance in a JTextField. Method update (lines 49–52) invokes method updateDisplay each time an Account notifies the AbstractAccountView of a change. Interface Observer defines method update, which takes as an Observable argument a reference to the Observable instance that issued the update. An Observable object issues an update by invoking method notifyObservers of class Observable. Method notifyObservers invokes method update for each registered Observer. An Observer that listens for updates from multiple Observable objects can use the Observable argument to determine which Observable object issued the update. The Object argument (line 50) contains optional data the Observable object may pass to an overloaded version of method notifyObservers. This Object could contain information about the specific data that changed in the model.

AccountTextView (Fig. 3.6) extends AbstractAccountView to provide a text-based view of Account data. Line 16 creates a JTextField in which AccountTextView displays the Account balance. Lines 19–20 create a NumberFormat field to format the Account balance as U. S. dollars. The AccountTextView constructor (lines 23–35) invokes the AbstractAccountView constructor with the given Account to perform required initialization (line 25). Line 28 makes the balanceTextField uneditable to prevent users from modifying the balance directly. Lines 31–32 add a JLabel and the balanceTextField to the AccountTextView. Line 34 invokes method updateDisplay to display the current Account balance.

```java
// AccountTextView.java
// AccountTextView is an AbstractAccountView subclass
// that displays an Account balance in a JTextField.
package com.deitel.advjhtp1.mvc.account;

// Java core packages
import java.util.*;
import java.text.NumberFormat;

// Java extension packages
import javax.swing.*;

public class AccountTextView extends AbstractAccountView {

    // JTextField for displaying Account balance
    private JTextField balanceTextField = new JTextField( 10 );

    // NumberFormat for US dollars
    private NumberFormat moneyFormat =
        NumberFormat.getCurrencyInstance( Locale.US );

    // AccountTextView constructor
    public AccountTextView( Account account )
    {
        super( account );
    }
}
```

Fig. 3.6 AccountTextView for displaying observed Account information in a JTextField (part 1 of 2).
Method *updateDisplay* (lines 38–43) implements abstract method *updateDisplay* of class *AbstractAccountView*. Lines 41–42 set the *balanceTextField*’s text to the formatted *Account* balance. Recall that method *update* of class *AbstractAccountView* invokes method *updateDisplay* each time method *update* receives a notification from the *Account* model.

*AccountBarGraphView* (Fig. 3.7) extends *AbstractAccountView* to provide a bar-graph view of *Account* data. Method *paintComponent* (lines 21–57) draws a bar graph for the current *Account* balance. Line 24 invokes method *paintComponent* of the superclass to follow the proper painting sequence. Line 27 gets the current *Account* balance. Line 32 calculates the length in pixels of the *Account*’s bar graph. The entire graph is 200 pixels wide and represents -$5,000 to +$5,000, so we divide the *Account* balance by $10,000 and multiply by 200 pixels to calculate the length of the the bar. If the *Account* balance is positive, lines 36–37 draw the bar graph in black. If the *Account* balance is negative, lines 42–43 draw the bar graph in red.
public class AccountBarGraphView extends AbstractAccountView {

    // AccountBarGraphView constructor
    public AccountBarGraphView( Account account ) {
        super( account );
    }

    // draw Account balance as a bar graph
    public void paintComponent( Graphics g ) {
        // ensure proper painting sequence
        super.paintComponent( g );

        // get Account balance
        double balance = getAccount().getBalance();

        // calculate integer height for bar graph (graph
        // is 200 pixels wide and represents Account balances
        // from -$5,000.00 to +$5,000.00)
        int barLength = ( int ) ( ( balance / 10000.0 ) * 200 );

        // if balance is positive, draw graph in black
        if ( balance >= 0.0 ) {
            g.setColor( Color.black );
            g.fillRect( 105, 15, barLength, 20 );
        }

        // if balance is negative, draw graph in red
        else {
            g.setColor( Color.red );
            g.fillRect( 105 + barLength, 15, -barLength, 20 );
        }

        // draw vertical and horizontal axes
        g.setColor( Color.black );
        g.drawLine( 5, 25, 205, 25 );
        g.drawLine( 105, 5, 105, 45 );

        // draw graph labels
        g.setFont( new Font( "SansSerif", Font.PLAIN, 10 ) );
        g.drawString( "-$5,000", 5, 10 );
        g.drawString( "$0", 110, 10 );
        g.drawString( "+$5,000", 166, 10 );

    } // end method paintComponent

    // repaint graph when display is updated
    public void updateDisplay() {
        repaint();
    }
}

Fig. 3.7 AccountBarGraphView for rendering observed Account information as a bar graph (part 2 of 3).
Method **updateDisplay** (lines 60–63) invokes method **repaint** (line 62) to update the bar graph’s display. **AbstractAccountView** method **update** invokes method **updateDisplay** each time the **Account** model notifies the view of a change in the model. Method **getPreferredSize** (lines 66–69) overrides method **getPreferredSize** of class **JPanel**. Line 68 returns a new **Dimension** object that specifies the **AccountBarGraphView**’s preferred size as 210 pixels wide by 50 pixels tall. Most **LayoutManager**s use method **getPreferredSize** to determine how much space to allocate for each component. Lines 72–81 override methods **getMinimumSize** and **getMaximumSize** to return the **AccountBarGraphView**’s preferred size.

**AssetPieChartView** (Fig. 3.8) provides a pie-chart view of multiple asset **Accounts**. **AssetPieChartView** defines method **addAccount** (line 25–42), which adds an **Account** to the List of **Accounts** shown in the pie chart. If the given **Account** reference is null, line 29 throws a **NullPointerException**. Otherwise, line 32 adds the **Account** to **accounts**. Line 35 invokes method **getRandomColor** and adds the random **Color** to the **colors Map**. **AssetPieChartView** uses this color to draw the **Account**’s wedge in the pie chart. The **Account** object itself is the **Color**’s key in the **Map**. Line 38 invokes method **addObserver** of class **Account** to register the **AssetPieChartView** for **Account** updates. Line 41 invokes method **repaint** the display the pie chart with the new **Account**’s information.

Method **removeAccount** (lines 45–58) removes an **Account** from the pie chart. Line 48 invokes method **deleteObserver** of class **Account** to unregister the **AssetPieChartView** as an **Observer** of the **Account**. Line 51 removes the **Account** from List **accounts**, and line 54 removes the **Account**’s color from **HashMap colors**. Line 57 invokes method **repaint** to update the pie-chart display.
// AssetPieChartView.java
// AssetPieChartView is an AbstractAccountView subclass that
displays multiple asset Account balances as a pie chart.
package com.deitel.advjhtp1.mvc.account;

import java.awt.*;
import java.util.*;
import java.util.List;
import javax.swing.*;
import javax.swing.border.*;

public class AssetPieChartView extends JPanel
    implements Observer {

private List accounts = new ArrayList();

private Map colors = new HashMap();

public void addAccount( Account account )
{
    // do not add null Accounts
    if ( account == null )
        throw new NullPointerException();

    // add Account to accounts Vector
    accounts.add( account );

    // add Color to Hashtable for drawing Account's wedge
    colors.put( account, getRandomColor() );

    // register as Observer to receive Account updates
    account.addObserver( this );

    // update display with new Account information
    repaint();
}

public void removeAccount( Account account )
{
    // stop receiving updates from given Account
    account.deleteObserver( this );

    // remove Account from accounts Vector
    accounts.remove( account );
// remove Account's Color from Hashtable
colors.remove( account );

// update display to remove Account information
repaint();
}

// draw Account balances in a pie chart
public void paintComponent( Graphics g )
{
    // ensure proper painting sequence
    super.paintComponent( g );

    // draw pie chart
drawPieChart( g );

    // draw legend to describe pie chart wedges
drawLegend( g );
}

// draw pie chart on given Graphics context
private void drawPieChart( Graphics g )
{
    // get combined Account balance
double totalBalance = getTotalBalance();

    // create temporary variables for pie chart calculations
double percentage = 0.0;
    int startAngle = 0;
    int arcAngle = 0;

    Iterator accountIterator = accounts.iterator();
    Account account = null;

    // draw pie wedge for each Account
    while ( accountIterator.hasNext() ) {

        // get next Account from Iterator
        account = ( Account ) accountIterator.next();

        // draw wedges only for included Accounts
        if ( !includeAccountInChart( account ) )
            continue;

        // get percentage of total balance held in Account
        percentage = account.getBalance() / totalBalance;

        // calculate arc angle for percentage
        arcAngle = ( int ) Math.round( percentage * 360 );

        // set drawing Color for Account pie wedge
        g.setColor( ( Color ) colors.get( account ) );

    }

Fig. 3.8 AssetPieChartView for rendering multiple observed asset Accounts as a pie chart (part 2 of 5).
Chapter 3 Model-View-Controller

105 // draw Account pie wedge
106 g.fillArc( 5, 5, 100, 100, startAngle, arcAngle );
107 // calculate startAngle for next pie wedge
108 startAngle += arcAngle;
109 }
110 } // end method drawPieChart
111
112 // draw pie chart legend on given Graphics context
113 private void drawLegend( Graphics g )
114 {
115     Iterator accountIterator = accounts.iterator();
116     Account account = null;
117
118     // create Font for Account name
119     Font font = new Font( "SansSerif", Font.BOLD, 12 );
120     g.setFont( font );
121
122     // get FontMetrics for calculating offsets and
123     // positioning descriptions
124     FontMetrics metrics = getFontMetrics( font );
125     int ascent = metrics.getMaxAscent();
126     int offsetY = ascent + 2;
127
128     // draw description for each Account
129     for ( int i = 1; accountIterator.hasNext(); i++ ) {
130         // get next Account from Iterator
131         account = ( Account ) accountIterator.next();
132
133         // draw Account color swatch at next offset
134         g.setColor( ( Color ) colors.get( account ) );
135         g.fillRect( 125, offsetY * i, ascent, ascent );
136
137         // draw Account name next to color swatch
138         g.setColor( Color.black );
139         g.drawString( account.getName(), 140,
140             offsetY * i + ascent );
141     }
142 } // end method drawLegend
143
144 // get combined balance of all observed Accounts
145 private double getTotalBalance()
146 {
147     double sum = 0.0;
148     Iterator accountIterator = accounts.iterator();
149     Account account = null;
150     for ( Account account : accountIterator ) {
151         sum += account.getBalance();
152     }
153     return sum;
154 } // end method getTotalBalance

Fig. 3.8 AssetPieChartView for rendering multiple observed asset Accounts as a pie chart (part 3 of 5).
// calculate total balance
while ( accountIterator.hasNext() ) {
    account = ( Account ) accountIterator.next();
    // add only included Accounts to sum
    if ( includeAccountInChart( account ) )
        sum += account.getBalance();
}
return sum;

// return true if given Account should be included in pie chart
protected boolean includeAccountInChart( Account account ) {
    // include only Asset accounts (Accounts with positive balances)
    return account.getBalance() > 0.0;
}

// get a random Color for drawing pie wedges
private Color getRandomColor() {
    // calculate random red, green and blue values
    int red = ( int ) ( Math.random() * 256 );
    int green = ( int ) ( Math.random() * 256 );
    int blue = ( int ) ( Math.random() * 256 );
    // return newly created Color
    return new Color( red, green, blue );
}

// receive updates from Observable Account
public void update( Observable observable, Object object ) {
    repaint();
}

// get AccountBarGraphView's preferred size
public Dimension getPreferredSize() {
    return new Dimension( 210, 110 );
}

// get AccountBarGraphView's preferred size
public Dimension getMinimumSize() {
    return getPreferredSize();
}

Fig. 3.8 AssetPieChartView for rendering multiple observed asset Accounts as a pie chart (part 4 of 5).
Method `paintComponent` (lines 61–71) invokes methods `drawPieChart` (line 67) and `drawLegend` (line 70) to draw the pie chart and chart legend, respectively. Method `drawPieChart` (lines 74–112) draws a pie-chart wedge for each `Account`. Line 77 invokes method `getTotalBalance` to get the total balance for all `Accounts`. Lines 80–111 calculate the percentage of the total balance held in each `Account` and draw the wedges. Line 91 gets the next `Account` from `accountIterator`. Line 94 invokes method `includeAccountInChart` to determine if the pie chart should include the current `Account`. If the chart should not include the `Account`, line 95 continues the `while` loop to the next iteration. Line 98 calculates the percentage of the total assets held in the current `Account`. Line 101 calculates the size of the `Account`’s pie wedge. Line 104 gets the `Account`’s color from `Map colors` and invokes method `setColor` of class `Graphics`. Line 107 invokes method `fillArc` of class `Graphics` to draw the `Account`’s pie wedge. The first four arguments to method `fillArc` specify the position and diameter of the arc, respectively. The third argument—`startAngle`—specifies the angle at which the arc should begin. The fourth argument—`arcAngle`—specifies the degrees of arc sweep. Line 101 sets the `startAngle` for the next pie wedge.

Method `drawLegend` (lines 115–145) draws a legend (shown in Fig. 3.10) to show which color represents each `Account`. The legend shows each color square and `Account` name in a list along the right side of the pie chart. Lines 137–138 set the `Font` in which to draw the `Account`. Lines 121–128 use a `FontMetrics` object to calculate the heights of characters in the current `Font`. Line 127 invokes method `getMaxAscent` of class `FontMetrics` to get the `maximum ascent` (i.e., maximum height above the baseline) of characters in the current `Font`. Line 128 calculates `offsetY` by adding 2 to the `Font`’s maximum ascent. We use `offsetY` to determine the position at which to draw each `Account`’s color square and name. Lines 131–144 draw the legend item for each `Account`. Line 134 gets the next `Account` from `accountIterator`. Lines 137–138 draw the color square, and lines 141–143 draw the `Account` name.

Method `getTotalBalance` (lines 148–165) calculates the total balance for all included `Accounts`. Line 160 invokes method `includeAccountInChart` to determine whether the calculation should include the current `Account`. If the calculation should include the `Account`, line 161 adds the `Account`'s balance to variable `sum`.

Method `includeAccountInChart` (lines 169–174) returns a `boolean` indicating whether the `Account` should be included in the pie chart. `AssetPieChartView` shows only asset `Accounts` (i.e., `Accounts` with positive balances). Line 173 returns `true` only if the `Account` balance is greater than zero. Subclasses can override this method to include and exclude `Accounts` based on other criteria.
Method `getRandomColor` (lines 177–186) generates a random `Color`. `AssetPieChartView` uses this method to generate a different `Color` for each `Account` in the pie chart. Lines 180–182 calculate random values for the `red`, `green` and `blue` `Color` components. Line 185 creates a new `Color` object using the random `red`, `green` and `blue` values and returns the new `Color` to the caller.

Method `update` (lines 189–192) invokes method `repaint` to update the pie-chart display. Method `getPreferredSize` (lines 195–198) returns the `AssetPieChartView`'s preferred size, which provides enough space to draw the pie chart and legend.

`AccountController` (Fig. 3.9) implements the controller in the MVC architecture. `AccountController` provides a user interface for modifying `Account` data. `AccountController` extends `JPanel` (line 14), because it provides a set of GUI components for depositing and withdrawing `Account` funds.

Line 28 sets the `account` member variable to the `Account` that `AccountController` will control. Line 31 creates a `JTextField` into which users can enter an amount to withdraw from, or deposit into, the controlled `Account`. Line 34 creates a `JButton` for depositing the given amount into the `Account`. The `depositButton`'s `ActionListener` (lines 37–55) invokes method `deposit` of class `Account` to deposit the amount entered in `amountTextField` (lines 44–45). If method `parseDouble` (line 44) throws a `NumberFormatException` because the text entered was not a valid number, lines 48–53 catch the exception and display an error message to the user.

```
// AccountController.java
// AccountController is a controller for Accounts. It provides
// a JTextField for inputting a deposit or withdrawal amount
// and JButtons for depositing or withdrawing funds.
package com.deitel.advjhtp1.mvc.account;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class AccountController extends JPanel {

    // Account to control
    private Account account;

    // JTextField for deposit or withdrawal amount
    private JTextField amountTextField;

    // AccountController constructor
    public AccountController( Account controlledAccount )
    {
        super();
    }
```

Fig. 3.9 `AccountController` for obtaining user input to modify `Account` information (part 1 of 3).
// account to control
account = controlledAccount;

// create JTextField for entering amount
amountTextField = new JTextField(10);

// create JButton for deposits
JButton depositButton = new JButton("Deposit");

depositButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        try {
            // deposit amount entered in amountTextField
            account.deposit(Double.parseDouble(amountTextField.getText()));
        }
        catch (NumberFormatException exception) {
            JOptionPane.showMessageDialog(AccountController.this,
            "Please enter a valid amount", "Error",
            JOptionPane.ERROR_MESSAGE);
        }
    }
});

// create JButton for withdrawals
JButton withdrawButton = new JButton("Withdraw");

withdrawButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        try {
            // withdraw amount entered in amountTextField
            account.withdraw(Double.parseDouble(amountTextField.getText()));
        }
        catch (NumberFormatException exception) {
            JOptionPane.showMessageDialog(AccountController.this,
            "Please enter a valid amount", "Error",
            JOptionPane.ERROR_MESSAGE);
        }
    }
});

Fig. 3.9 AccountController for obtaining user input to modify Account information (part 2 of 3).
104 Model-View-Controller Chapter 3

Line 59 creates a JButton for withdrawing the given amount from the Account. The withdrawButton’s ActionListener (lines 62–80) invokes method withdraw of class Account to withdraw the amount entered in amountTextField (lines 69–70). If method parseDouble (line 69) throws a NumberFormatException, because the text entered was not a valid number, lines 73–78 catch the exception and display an error message to the user. Lines 84–88 lay out amountTextField, a JLabel, depositButton and withdrawButton.

AccountManager (Fig. 3.10) is an application that uses MVC to manage Account information. Lines 22 creates a new Account with the name Account1 and a $1,000.00 balance. Line 25 invokes method getAccountPanel of class AccountManager to create a JPanel containing view and controller components for account1. Line 28 creates a new Account with the name Account2 and a $3,000.00 balance. Line 31 invokes method createAccountPanel to create a JPanel containing view and controller components for account2. Lines 34–35 create an AssetPieChartView for displaying account1 and account2 information in a pie chart. Lines 38–39 invoke method addAccount of class AssetPieChartView to add account1 and account2 to the pie chart. Lines 42–47 create a JPanel with a TitledBorder for the AssetPieChartView. Lines 50–54 lay out the JPanels for each account and AssetPieChartView.


Figure 3.10 shows sample AccountManager output. Notice as you run the program that the views reflect each withdrawal or deposit immediately. For example, depositing 1500.00 in Account1 causes the AccountTextView for Account1 to display $2,500.00, the AccountBarGraphView for Account1 to display a larger bar graph and AssetPieChartView to display a larger wedge for Account1. Withdrawing 4623.12 from Account2 causes a new balance of ($1,623.12) (parentheses
indicate a negative balance) to be shown, a red bar graph to be displayed and the **Account** 2 wedge from **AssetPieChartView** to be removed. If both **Accounts** have negative balances, **AssetPieChartView** removes both **Accounts** from the pie chart.

```java
// AccountManager.java
// AccountManager is an application that uses the MVC design
// pattern to manage bank Account information.
package com.deitel.advjhtp1.mvc.account;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;
import javax.swing.border.*;

public class AccountManager extends JFrame {

    // AccountManager no-argument constructor
    public AccountManager() {
        super( "Account Manager" );

        // create account1 with initial balance
        Account account1 = new Account( "Account 1", 1000.00 );

        // create GUI for account1
        JPanel account1Panel = createAccountPanel( account1 );

        // create account2 with initial balance
        Account account2 = new Account( "Account 2", 3000.00 );

        // create GUI for account2
        JPanel account2Panel = createAccountPanel( account2 );

        // create AccountPieChartView to show Account pie chart
        AssetPieChartView pieChartView =
            new AssetPieChartView();

        // add both Accounts to AccountPieChartView
        pieChartView.addAccount( account1 );
        pieChartView.addAccount( account2 );

        // create JPanel for AccountPieChartView
        JPanel pieChartPanel = new JPanel();

        pieChartPanel.setBorder(
            new TitledBorder( "Assets" ) );
    }

    // method for creating AccountGUI
    private JPanel createAccountPanel( Account account ) {
        // Create Panel for account
        JPanel accountPanel = new JPanel();

        // Create Account label
        JLabel accountLabel = new JLabel( account.getName() );

        // Create Account Balance label
        JLabel balanceLabel = new JLabel( account.getBalance() );

        // Create static text
        JLabel staticText = new JLabel( "Account" );

        // Add components to accountPanel
        accountPanel.add( accountLabel );
        accountPanel.add( balanceLabel );
        accountPanel.add( staticText );

        return accountPanel;
    }

    public static void main( String[] args ) {
        new AccountManager();
    }
}
```

**Fig. 3.10** **AccountManager** application for displaying and modifying **Account** information using the model-view-controller architecture (part 1 of 3).
Fig. 3.10  AccountManager application for displaying and modifying Account information using the model-view-controller architecture (part 2 of 3).

```java
    pieChartPanel.add( pieChartView );

    // lay out account1, account2 and pie chart components
    Container contentPane = getContentPane();
    contentPane.setLayout( new GridLayout( 3, 1 ) );
    contentPane.add( account1Panel );
    contentPane.add( account2Panel );
    contentPane.add( pieChartPanel );

    setSize( 425, 450 );

} // end AccountManager constructor

// create GUI components for given Account
private JPanel createAccountPanel( Account account )
{
    // create JPanel for Account GUI
    JPanel accountPanel = new JPanel();

    // set JPanel's border to show Account name
    accountPanel.setBorder( new TitledBorder( account.getName() ) );

    // create AccountTextView for Account
    AccountTextView accountTextView = new AccountTextView( account );

    // create AccountBarGraphView for Account
    AccountBarGraphView accountBarGraphView = new AccountBarGraphView( account );

    // create AccountController for Account
    AccountController accountController = new AccountController( account );

    // lay out Account's components
    accountPanel.add( accountController );
    accountPanel.add( accountTextView );
    accountPanel.add( accountBarGraphView );

    return accountPanel;

} // end method getAccountPanel

// execute application
public static void main( String args[] )
{
    AccountManager manager = new AccountManager();
    manager.setDefaultCloseOperation( EXIT_ON_CLOSE );
    manager.setVisible( true );
}
```
3.4 JList

JList is a Swing component that implements the delegate-model architecture. JList acts as a delegate for an underlying ListModel (Fig. 3.11). Interface ListModel defines methods for getting list elements, getting the size of the list and registering and unregistering ListDataListeners. A ListModel notifies each registered ListDataListener of each change in the ListModel.

Class PhilosophersJList (Fig. 3.12) uses a JList and DefaultListModel to display a list of philosophers. Class DefaultListModel provides a basic ListModel implementation. Line 23 creates a new DefaultListModel, and lines 24–31 add several philosophers to the DefaultListModel. Line 34 creates a new JList and
passes the philosophers DefaultListModel to the JList constructor. The JList constructor registers the JList as a ListDataListener of the DefaultListModel, so that updates to the DefaultListModel will be reflected in the JList. Lines 37–38 set the JList's selection mode to allow the user to select only one philosopher at a time. The selection modes are constant integer values defined in interface ListSelectionModel. For example, MULTIPLE_INTERVAL_SELECTION allows the user to select multiple, separate intervals in the JList.

```java
// PhilosophersJList.java
// MVC architecture using JList with a DefaultListModel
package com.deitel.advjhtp1.mvc.list;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class PhilosophersJList extends JFrame {
    private DefaultListModel philosophers;
    private JList list;

    // PhilosophersJList constructor
    public PhilosophersJList() {
        super( "Favorite Philosophers" );

        // create a DefaultListModel to store philosophers
        philosophers = new DefaultListModel();
        philosophers.addElement( "Socrates" );
        philosophers.addElement( "Plato" );
        philosophers.addElement( "Aristotle" );
        philosophers.addElement( "St. Thomas Aquinas" );
        philosophers.addElement( "Soren Kierkegaard" );
        philosophers.addElement( "Immanuel Kant" );
        philosophers.addElement( "Friedrich Nietzsche" );
        philosophers.addElement( "Hannah Arendt" );
```
// create a JList for philosophers DefaultListModel
list = new JList( philosophers );

// allow user to select only one philosopher at a time
list.setSelectionMode(
    ListSelectionModel.SINGLE_SELECTION );

// create JButton for adding philosophers
JButton addButton = new JButton( "Add Philosopher" );
addButton.addActionListener(
    new ActionListener() {

        public void actionPerformed( ActionEvent event ) {
            // prompt user for new philosopher's name
            String name = JOptionPane.showInputDialog(
                PhilosophersJList.this, "Enter Name" );

            // add new philosopher to model
            philosophers.addElement( name );
        }
    });

// create JButton for removing selected philosopher
JButton removeButton =
    new JButton( "Remove Selected Philosopher" );
removeButton.addActionListener(
    new ActionListener() {

        public void actionPerformed( ActionEvent event ) {
            // remove selected philosopher from model
            philosophers.removeElement( list.getSelectedValue() );
        }
    });

// lay out GUI components
JPanel inputPanel = new JPanel();
inputPanel.add( addButton );
inputPanel.add( removeButton );

Container container = getContentPane();
container.add( list, BorderLayout.CENTER );
container.add( inputPanel, BorderLayout.NORTH );

setDefaultCloseOperation( EXIT_ON_CLOSE );
setSize( 400, 300 );
setVisible( true );
Lines 41–55 create a JButton for adding new philosophers to the DefaultListModel. Lines 48–49 in method actionPerformed invoke static method showInputDialog of class JOptionPane to prompt the user for the philosopher’s name.
Line 52 invokes method `addElement` of class `DefaultListModel` to add the new philosopher to the list. The `DefaultListModel` will notify the `JList` that the model changed, and the `JList` will update the display to include the new list item.

Lines 58–71 create a `JButton` for deleting a philosopher from the `DefaultListModel`. Lines 67–68 in method `actionPerformed` invoke method `getSelectedValue` of class `JList` to get the currently selected philosopher and invoke method `removeElement` of class `DefaultListModel` to remove the philosopher. The `DefaultListModel` will notify the `JList` that the model changed, and the `JList` will update the display to remove the deleted philosopher. Lines 74–84 lay out the GUI components and set `JFrame` properties for the application window.

### 3.5 JTable

`JTable` is another `Swing` component that implements the delegate-model architecture. `JTables` are delegates for tabular data stored in `TableModel` implementations. Interface `TableModel` declares methods for retrieving and modifying data (e.g., the value in a certain table cell) and for retrieving and modifying metadata (e.g., the number of columns and rows). The `JTable` delegate invokes `TableModel` methods to build its view of the `TableModel` and to modify the `TableModel` based on user input.

Figure 3.13 describes the methods defined in interface `TableModel`. Custom implementations of interface `TableModel` can use arbitrary internal representations of the tabular data. For example, the `DefaultTableModel` implementation uses `Vectors` to store the rows and columns of data. In Chapter 8, JDBC, we implement interface `TableModel` to create a `TableModel` that represents data stored in a JDBC `ResultSet`. Figure 3.14 illustrates the delegate-model relationship between `JTable` and `TableModel`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addTableModelListener(TableModelListener listener)</td>
<td>Add a <code>TableModelListener</code> to the <code>TableModel</code>. The <code>TableModel</code> will notify the <code>TableModelListener</code> of changes in the <code>TableModel</code>.</td>
</tr>
<tr>
<td>void removeTableModelListener(TableModelListener listener)</td>
<td>Remove a previously added <code>TableModelListener</code> from the <code>TableModel</code>.</td>
</tr>
<tr>
<td>Class getColumnClass(int columnIndex)</td>
<td>Get the <code>Class</code> object for values in the column with specified <code>columnIndex</code>.</td>
</tr>
<tr>
<td>int getColumnCount()</td>
<td>Get the number of columns in the <code>TableModel</code>.</td>
</tr>
<tr>
<td>String getColumnName(int columnIndex)</td>
<td>Get the name of the column with the given <code>columnIndex</code>.</td>
</tr>
<tr>
<td>int getRowCount()</td>
<td>Get the number of rows in the <code>TableModel</code>.</td>
</tr>
</tbody>
</table>
PhilosophersJTable (Fig. 3.15) displays philosopher information in a JTable using a DefaultTableModel. Class DefaultTableModel implements interface TableModel and uses Vectors to represent the rows and columns of data. Line 24 creates the philosophers DefaultTableModel. Lines 27–29 add columns to the DefaultTableModel for the philosophers’ first names, last names and years in which they lived. Lines 32–53 create rows for seven philosophers. Each row is a String array whose elements are the philosopher’s first name, last name and the year in which the philosopher lived, respectively. Method addRow of class DefaultTableModel adds each philosopher to the DefaultTableModel. Line 56 creates the JTable that will act as a delegate for the philosophers DefaultTableModel.

Lines 59–72 create a JButton and ActionListener for adding a new philosopher to the DefaultTableModel. Line 66 in method actionPerformed creates a String array of three empty elements. Line 69 adds the empty String array to the DefaultTableModel. This causes the JTable to display a blank row at the bottom of the JTable. The user can then type the philosopher’s information directly into the JTable cells. This demonstrates the JTable delegate acting as a controller, because it modifies the DefaultTableModel based on user input.
// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;
import javax.swing.table.*;

public class PhilosophersJTable extends JFrame {

    private DefaultTableModel philosophers;
    private JTable table;

    // PhilosophersJTable constructor
    public PhilosophersJTable()
    {
        super( "Favorite Philosophers" );

        // create a DefaultTableModel to store philosophers
        philosophers = new DefaultTableModel();

        // add Columns to DefaultTableModel
        philosophers.addColumn( "First Name" );
        philosophers.addColumn( "Last Name" );
        philosophers.addColumn( "Years" );

        // add philosopher names and dates to DefaultTableModel
        String[] socrates = { "Socrates", "", "469-399 B.C." };
        philosophers.addRow( socrates );

        String[] plato = { "Plato", "", "428-347 B.C." };
        philosophers.addRow( plato );

        String[] aquinas = { "Thomas", "Aquinas", "1225-1274" };
        philosophers.addRow( aquinas );

        String[] kierkegaard = { "Soren", "Kierkegaard", "1813-1855" };
        philosophers.addRow( kierkegaard );

        String[] kant = { "Immanuel", "Kant", "1724-1804" };
        philosophers.addRow( kant );

        String[] nietzsche = { "Friedrich", "Nietzsche", "1844-1900" };
        philosophers.addRow( nietzsche );

        String[] arendt = { "Hannah", "Arendt", "1906-1975" };
        philosophers.addRow( arendt );
    }
}

Fig. 3.15 PhilosophersJTable application demonstrating JTable and DefaultTableModel (part 2 of 4).
// create a JTable for philosophers DefaultTableModel
table = new JTable( philosophers );

// create JButton for adding philosophers
JButton addButton = new JButton( 'Add Philosopher' );
addButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // create empty array for new philosopher row
            String[] philosopher = { '', '', '' }; // add empty philosopher row to model
            philosophers.addRow( philosopher );
        }
    });

// create JButton for removing selected philosopher
JButton removeButton = new JButton( 'Remove Selected Philosopher' );
removeButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // remove selected philosopher from model
            philosophers.removeRow( table.getSelectedRow() );
        }
    });

// lay out GUI components
JPanel inputPanel = new JPanel();
inputPanel.add( addButton );
inputPanel.add( removeButton );
Container container = getContentPane();
container.add( new JScrollPane( table ), BorderLayout.CENTER );
container.add( inputPanel, BorderLayout.NORTH );
setDefaultCloseOperation( EXIT_ON_CLOSE );
setSize( 400, 300 );
setVisible( true );

} // end PhilosophersJTable constructor

Fig. 3.15 PhilosophersJTable application demonstrating JTable and
DefaultTableModel (part 3 of 4).
Lines 75–88 create a JButton and ActionListener for removing a philosopher from the DefaultTableModel. Lines 84–85 in method actionPerformed retrieve the currently selected row in the JTable delegate and invoke method removeRow of class DefaultTableModel to remove the selected row. The DefaultTableModel notifies the JTable that the DefaultTableModel has changed, and the JTable removes the appropriate row from the display. Lines 96–97 add the JTable to a JScrollPane. JTables will not display their column headings unless placed within a JScrollPane.

### 3.6 JTree

JTree is one of the more complex Swing components that implements the delegate-model architecture. TreeModels represent hierarchical data, such as family trees, certain types of file systems, company management structures and document outlines. JTrees act as delegates (i.e., combined view and controller) for TreeModels.
To describe tree data structures, it is common to use terms that more commonly describe family trees. A tree data structure consists of a set of nodes (i.e., members or elements of the tree) that are related as parents, children, siblings, ancestors and descendents. A parent is a node that has other nodes as its children. A child is a node that has a parent. Sibling nodes are two or more nodes that share the same parent. An ancestor is a node that has children that also have children. A descendent is a node whose parent also has a parent. A tree must have one node—called the root node—that is the parent or ancestor of all other nodes in the tree. [Note: Unlike in a family tree, in a tree data structure a child node can have only one parent.]

Figure 3.16 shows the relationships among nodes in a tree. The **JT**ree contains a hierarchy of philosophers whose root is node Philosophers. Node Philosophers has seven child nodes, representing the major eras of philosophy—Ancient, Medieval, Renaissance, Early Modern, Enlightenment, 19th Century and 20th Century. Each philosopher (e.g., Socrates, St. Thomas Aquinas and Immanuel Kant) is a child of the philosopher’s era and is a descendent of node Philosophers. Nodes Socrates, Plato and Aristotle are sibling nodes, because they share the same parent node (Ancient).

---

3. Note that nodes in the tree data structures we discuss in this section each have only a single parent, unlike a family tree.
3.6.1 Using DefaultTreeModel

Interface **TreeModel** declares methods for representing a tree data structure in a **JTree**. Objects of any class can represent nodes in a **TreeModel**. For example, a **Person** class could represent a node in a family tree **TreeModel**. Class **DefaultTreeModel** provides a default **TreeModel** implementation. Interface **TreeNode** defines common operations for nodes in a **DefaultTreeModel**, such as **getParent** and **getAllowChildren**. Interface **MutableTreeNode** extends interface **TreeNode** to represent a node that can change, either by adding or removing child nodes or by changing the **Object** associated with the node. Class **DefaultMutableTreeNode** provides a **MutableTreeNode** implementation suitable for use in a **DefaultTreeModel**.

**Software Engineering Observation 3.3**

Although a **TreeModel** implementation can use objects of any class to represent the **TreeModel**’s nodes, the **TreeModel** implementation must be able to determine the hierarchical relationships among those objects. For example, a **Person** class would have to provide methods such as **getParent** and **getChildren** for use in a family tree **TreeModel**.

**JTree** employs two interfaces to implement the **JTree**’s delegate functionality. Interface **TreeCellRenderer** represents an object that creates a view for each node in the **JTree**. Class **DefaultTreeCellRenderer** implements interface **TreeCellRenderer** and extends class **JLabel** to provide a **TreeCellRenderer** default implementation. Interface **TreeCellEditor** represents an object for controlling (i.e., editing) each node in the **JTree**. Class **DefaultTreeCellEditor** implements interface **TreeCellEditor** and uses a **JTextField** for the **TreeCellEditor** default implementation.

**PhilosophersJTree** (Fig. 3.17) uses a **DefaultTreeModel** to represent a set of philosophers. The **DefaultTreeModel** organizes the philosophers hierarchically according to their associated eras in the history of philosophy. Lines 26–27 invoke method **createPhilosopherTree** to get the root, **DefaultMutableTreeNode**, which contains all the philosopher nodes. Line 30 creates a **DefaultTreeModel** and passes the **philosophersNode** **DefaultMutableTreeNode** to the **DefaultTreeModel** constructor. Line 33 creates a **JTree** and passes **DefaultTreeModel philosophers** to the **JTree** constructor.

```
// PhilosophersJTree.java
// MVC architecture using JTree with a DefaultTreeModel
package com.deitel.advjhtp1.mvc.tree;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.awt.util.*;

// Java extension packages
import javax.swing.*;
import javax.swing.tree.*;
```

**Fig. 3.17** **PhilosophersJTree** application demonstrating **JTree** and **DefaultTreeModel** (part 1 of 6).
public class PhilosophersJTree extends JFrame {

    private JTree tree;
    private DefaultTreeModel philosophers;
    private DefaultMutableTreeNode rootNode;

    // PhilosophersJTree constructor
    public PhilosophersJTree() {
        super( "Favorite Philosophers" );

        // get tree of philosopher DefaultMutableTreeNodes
        DefaultMutableTreeNode philosophersNode =
            createPhilosopherTree();

        // create philosophers DefaultTreeModel
        philosophers = new DefaultTreeModel( philosophersNode );

        // create JTree for philosophers DefaultTreeModel
        tree = new JTree( philosophers );

        // create JButton for adding philosophers
        JButton addButton = new JButton( "Add" );
        addButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent event ) {
                    addElement();
                }
            });

        // create JButton for removing selected philosopher
        JButton removeButton =
            new JButton( "Remove" );
        removeButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent event ) {
                    removeElement();
                }
            });

        // lay out GUI components
        JPanel inputPanel = new JPanel();
        inputPanel.add( addButton );
        inputPanel.add( removeButton );
    }

    // PhilosophersJTree application demonstrating JTree and DefaultTreeModel (part 2 of 6).
Container container = getContentPane();

container.add( new JScrollPane( tree ),
        BorderLayout.CENTER );

container.add( inputPanel, BorderLayout.NORTH );

setDefaultCloseOperation( EXIT_ON_CLOSE );
setSize( 400, 300 );
setVisible( true );

} // end PhilosophersJTree constructor

// add new philosopher to selected era
private void addElement()
{
    // get selected era
    DefaultMutableTreeNode parent = getSelectedNode();

    // ensure user selected era first
    if ( parent == null ) {
        JOptionPane.showMessageDialog( PhilosophersJTree.this, "Select an era.",
        "Error", JOptionPane.ERROR_MESSAGE );

        return;
    }

    // prompt user for philosopher's name
    String name = JOptionPane.showInputDialog( PhilosophersJTree.this, "Enter Name:" );

    // add new philosopher to selected era
    philosophers.insertNodeInto( new DefaultMutableTreeNode( name ),
        parent, parent.getChildCount() );

} // end method addElement

// remove currently selected philosopher
private void removeElement()
{
    // get selected node
    DefaultMutableTreeNode selectedNode = getSelectedNode();

    // remove selectedNode from model
    if ( selectedNode != null )
        philosophers.removeNodeFromParent( selectedNode );

} // end method removeElement

Fig. 3.17 PhilosophersJTree application demonstrating JTree and DefaultTreeModel (part 3 of 6).
private DefaultMutableTreeNode getSelectedNode()
{
    return ( DefaultMutableTreeNode )
    tree.getLastSelectedPathComponent();
}

private DefaultMutableTreeNode createPhilosopherTree()
{
    DefaultMutableTreeNode rootNode =
    new DefaultMutableTreeNode( "Philosophers" );

    DefaultMutableTreeNode ancient =
    new DefaultMutableTreeNode( "Ancient" );
    rootNode.add( ancient );

    ancient.add( new DefaultMutableTreeNode( "Socrates" ) );
    ancient.add( new DefaultMutableTreeNode( "Plato" ) );
    ancient.add( new DefaultMutableTreeNode( "Aristotle" ) );

    DefaultMutableTreeNode medieval =
    new DefaultMutableTreeNode( "Medieval" );
    rootNode.add( medieval );

    medieval.add( new DefaultMutableTreeNode( "St. Thomas Aquinas" ) );

    DefaultMutableTreeNode renaissance =
    new DefaultMutableTreeNode( "Renaissance" );
    rootNode.add( renaissance );

    renaissance.add( new DefaultMutableTreeNode( "Thomas More" ) );

    DefaultMutableTreeNode earlyModern =
    new DefaultMutableTreeNode( "Early Modern" );
    rootNode.add( earlyModern );

    earlyModern.add( new DefaultMutableTreeNode( "John Locke" ) );

    DefaultMutableTreeNode enlightenment =
    new DefaultMutableTreeNode( "Enlightenment" );
    rootNode.add( enlightenment );

Fig. 3.17 PhilosophersJTree application demonstrating JTree and DefaultTreeModel (part 4 of 6).
enlightenment.add( new DefaultMutableTreeNode("Immanuel Kant") );

// 19th Century Philosophers
DefaultMutableTreeNode nineteenth =
    new DefaultMutableTreeNode("19th Century");
rootNode.add( nineteenth );
nineteenth.add( new DefaultMutableTreeNode("Soren Kierkegaard") );
nineteenth.add( new DefaultMutableTreeNode("Friedrich Nietzsche") );

// 20th Century Philosophers
DefaultMutableTreeNode twentieth =
    new DefaultMutableTreeNode("20th Century");
rootNode.add( twentieth );
twentieth.add( new DefaultMutableTreeNode("Hannah Arendt") );

return rootNode;

} // end method createPhilosopherTree

// execute application
public static void main( String args[] )
{
    new PhilosophersJTree();
}

Fig. 3.17 PhilosophersJTree application demonstrating JTree and
DefaultTreeModel (part 5 of 6).
Lines 36–45 create a JButton and an ActionListener for adding a philosopher to the philosophers DefaultTreeModel. Line 42 in method actionPerformed invokes method addElement to add a new philosopher. Lines 48–59 create a JButton and an ActionListener for removing a philosopher from the philosophers DefaultTreeModel. Line 56 invokes method removeElement to remove the currently selected philosopher from the model.

Method addElement (lines 80–103) gets the currently selected node in the JTree by invoking method getSelectedNode (line 83). Method addElement inserts the new philosopher node as a child of the currently selected node. If there is no node currently selected, line 91 returns from method addElement without adding a new node. Lines 95–96 invoke static method showInputDialog of class JOptionPane to prompt the user for the new philosopher’s name. Lines 99–101 invoke method insertNodeInto of class DefaultTreeModel to insert the new philosopher in the model. Line 100 creates a new DefaultMutableTreeNode for the given philosopher. Line 101 specifies the parent node to which the new philosopher should be added. The final argument to method insertNodeInto specifies the index at which the new node should be inserted. Line 101 invokes method getChildCount of class DefaultMutableTreeNode to get the total number of children in node parent, which will cause the new node to be added as the last child of parent.
Method **removeElement** (lines 106–114) invokes method **getSelectedNode** (line 109) to get the currently selected node in the **JTree**. If **selectedNode** is not **null**, line 113 invokes method **removeNodeFromParent** of class **DefaultTreeModel** to remove **selectedNode** from the model. Method **getSelectedNode** (lines 117–122) invokes method **getSelectedPathComponent** of class **JTree** to get a reference to the currently selected node (line 121). Line 120 casts the selected node to **DefaultMutableTreeNode** and returns the reference to the caller.

Method **createPhilosopherTree** (lines 125–192) creates **DefaultMutableTreeNode**s for several philosophers and for the eras in which the philosophers lived. Lines 128–129 create a **DefaultMutableTreeNode** for the tree’s root. Class **DefaultMutableTreeNode** has property **userObject** that stores an **Object** that contains the node’s data. The **String** passed to the **DefaultMutableTreeNode** constructor (line 129) is the **userObject** for **rootNode**. The **JTree**’s **TreeCellRenderer** will invoke method **toString** of class **DefaultMutableTreeNode** to get a **String** to display for this node in the **JTree**.

Software Engineering Observation 3.4

Method **toString** of class **DefaultMutableTreeNode** returns the value returned by its **userObject**’s **toString** method.

Lines 132–134 create a **DefaultMutableTreeNode** for the **ancient** era of philosophy and add node **ancient** as a child of **rootNode** (line 134). Lines 136–138 create **DefaultMutableTreeNode**s for three ancient philosophers and add each **DefaultMutableTreeNode** as a child of **DefaultMutableTreeNode** **ancient**. Lines 141–189 create several additional **DefaultMutableTreeNode**s for other eras in the history of philosophy and for philosophers in those eras. Line 191 returns **rootNode**, which now contains the era and philosopher **DefaultMutableTreeNode**s as its children and descendants, respectively.

### 3.6.2 Custom **TreeModel** Implementation

If the **DefaultTreeModel** implementation is not sufficient for an application, developers also can provide custom implementations of interface **TreeModel**. **FileSystemModel** (Fig. 3.18) implements interface **TreeModel** to provide a model of a computer’s file system. A file system consists of directories and files arranged in a hierarchy. Line 17 declares a **File** reference **root** that serves as the root node in the hierarchy. This **File** is a directory that contains files and other directories. The **FileSystemModel** constructor (lines 23–26) takes a **File** argument for the **FileSystemModel** root. Method **getRoot** (lines 29–32) returns the **FileSystemModel**’s root node.

```java
// FileSystemModel.java
// TreeModel implementation using File objects as tree nodes.
package com.deitel.advjhtp1.mvc.tree.filesystem;

// Java core packages
import java.io.*;
```

**Fig. 3.18** **FileSystemModel** implementation of interface **TreeModel** to represent a file system (part 1 of 5).
import java.util.*;

// Java extension packages
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

public class FileSystemModel implements TreeModel {

    // hierarchy root
    private File root;

    // TreeModelListeners
    private Vector listeners = new Vector();

    // FileSystemModel constructor
    public FileSystemModel( File rootDirectory )
    {
        root = rootDirectory;
    }

    // get hierarchy root (root directory)
    public Object getRoot() {
        return root;
    }

    // get parent's child at given index
    public Object getChild( Object parent, int index ) {
        // get parent File object
        File directory = ( File ) parent;

        // get list of files in parent directory
        String[] children = directory.list();

        // return File at given index and override toString
        // method to return only the File's name
        return new TreeFile( directory, children[ index ] );
    }

    // get parent's number of children
    public int getChildCount( Object parent ) {
        // get parent File object
        File file = ( File ) parent;

        // get number of files in directory
        if ( file.isDirectory() ) {
            String[] fileList = file.list();
        }

    }

    Fig. 3.18  FileSystemModel implementation of interface TreeModel to represent a file system (part 2 of 5).
```java
if ( fileList != null )
    return fileList.list().length;

return 0; // childCount is 0 for files

// return true if node is a file, false if it is a directory
public boolean isLeaf( Object node )
{
    File file = ( File ) node;
    return file.isFile();
}

// get numeric index of given child node
public int getIndexOfChild( Object parent, Object child )
{
    // get parent File object
    File directory = ( File ) parent;

    // get child File object
    File file = ( File ) child;

    // get File list in directory
    String[] children = directory.list();

    // search File list for given child
    for ( int i = 0; i < children.length; i++ ) {
        if ( file.getName().equals( children[ i ] ) ) {
            // return matching File's index
            return i;
        }
    }

    return -1; // indicate child index not found
}

// invoked by delegate if value of Object at given
// TreePath changes
public void valueForPathChanged( TreePath path, Object value )
{
    // get File object that was changed
    File oldFile = ( File ) path.getLastPathComponent();

    // get parent directory of changed File
    String fileParentPath = oldFile.getParent();

    // FileSystemModel implementation of interface TreeModel to
    // represent a file system (part 3 of 5).
```
// get value of newFileName entered by user
String newFileName = (String) value;

// create File object with newFileName for renaming oldFile
File targetFile = new File(
    fileParentPath, newFileName);

// rename oldFile to targetFile
oldFile.renameTo(targetFile);

// get File object for parent directory
File parent = new File(fileParentPath);

// create int array for renamed File's index
int[] changedChildrenIndices =
    {getIndexOfChild(parent, targetFile)};

// create Object array containing only renamed File
Object[] changedChildren = {targetFile};

// notify TreeModelListeners of node change
fireTreeNodesChanged(path.getParentPath(),
    changedChildrenIndices, changedChildren);

}  // end method valueForPathChanged

// notify TreeModelListeners that children of parent at given TreePath with given indices were changed
private void fireTreeNodesChanged(TreePath parentPath,
    int[] indices, Object[] children)
{
    // create TreeModelEvent to indicate node change
    TreeModelEvent event = new TreeModelEvent(this,
        parentPath, indices, children);

    Iterator iterator = listeners.iterator();
    TreeModelListener listener = null;

    // send TreeModelEvent to each listener
    while (iterator.hasNext()) {
        listener = (TreeModelListener) iterator.next();
        listener.treeNodesChanged(event);
    }
}  // end method fireTreeNodesChanged

// add given TreeModelListener
public void addTreeModelListener(
    TreeModelListener listener)
{
    listeners.add(listener);
}
When building its view of a `TreeModel`, a `JTree` repeatedly invokes method `get-child` (lines 35–46) to traverse the `TreeModel`’s nodes. Method `get-child` returns argument `parent`’s child node at the given `index`. The nodes in a `TreeModel` need not implement interface `TreeNode` or interface `MutableTreeNode`; any `Object` can be a node in a `TreeModel`. In class `FileSystemModel`, each node is a `File`. Line 38 casts `Object` reference `parent` to a `File` reference. Line 41 invokes method `list` of class `File` to get a list of file names in the directory. Line 45 returns a new `TreeFile` object for the `File` at the given `index`. `JTree` invokes method `toString` of class `TreeFile` to get a label for the node in the `JTree`.

Method `get-child-count` (lines 49–64) returns the number of children contained in argument `parent`. Line 52 casts `Object` reference `parent` to a `File` reference named `file`. If `file` is a directory (line 55), lines 57–60 get a list of file names in the directory and return the `length` of the list. If `file` is not a directory, line 63 returns `0`, to indicate that `file` has no children.

A `JTree` invokes method `is-leaf` of class `FileSystemModel` (lines 67–71) to determine if `Object` argument `node` is a `leaf node`—a node that does not contain children. In a file system, only directories can contain children, so line 70 returns `true` only if argument `node` is a file (not a directory).

---

4. Leaf node controls the initial screen display of the expand handle.
Method `getIndexOfChild` (lines 74–98) returns argument `child`’s index in the given `parent` node. For example, if `child` were the third node in `parent`, method `getIndexOfChild` would return zero-based index 2. Lines 77 and 80 get `File` references for the `parent` and `child` nodes, respectively. Line 83 gets a list of files, and lines 86–93 search through the list for the given `child`. If the filename in the list matches the given child (line 88), line 91 returns the index `i`. Otherwise, line 95 returns `-1`, to indicate that `parent` did not contain `child`.

The `JTree` delegate invokes method `valueForPathChanged` (lines 101–135) when the user edits a node in the tree. A user can click on a node in the `JTree` and edit the node’s name, which corresponds to the associated `File` object’s file name. When a user edits a node, `JTree` invokes method `valueForPathChanged` and passes a `TreePath` argument that represents the changed node’s location in the tree, and an `Object` that contains the node’s new value. In this example, the new value is a new file name `String` for the associated `File` object. Line 105 invokes method `getLastPathComponent` of class `TreePath` to obtain the `File` object to rename. Line 108 gets `oldFile`’s parent directory. Line 111 casts argument `value`, which contains the new file name, to a `String`. Lines 115–116 create `File` object `targetFile` using the new file name. Line 119 invokes method `renameTo` of class `File` to rename `oldFile` to `targetFile`.

After renaming the file, the `FileSystemModel` must notify its `TreeModelListeners` of the change by issuing a `TreeModelEvent`. A `TreeModelEvent` that indicates a node change includes a reference to the `TreeModel` that generated the event, the `TreePath` of the changed nodes’ parent node, an integer array containing the changed nodes’ indices and an `Object` array containing references to the changed nodes themselves. Line 122 creates a `File` object for the renamed file’s parent directory. Lines 125–126 create an integer array for the indices of changed nodes. Line 128 creates an `Object` array of changed nodes. The integer and `Object` arrays have only one element each because only one node changed. If multiple nodes were changed, these arrays would need to include elements for each changed node. Lines 132–133 invoke method `fireTreeNodesChanged` to issue the `TreeModelEvent`.

**Performance Tip 3.1**

`JTree` uses the index and `Object` arrays in a `TreeModelEvent` to determine which nodes in the `JTree` need to be updated. This method improves performance by updating only the nodes that have changed, and not the entire `JTree`.

Method `fireTreeNodesChanged` (lines 139–154) issues a `TreeModelEvent` to all registered `TreeModelListeners`, indicating that nodes in the `TreeModel` have changed. `TreePath` argument `parentPath` is the path to the parent whose child nodes changed. The integer and `Object` array arguments contain the indices of the changed nodes and references to the changed nodes, respectively. Lines 143–144 create the `TreeModel` event with the given event data. Lines 150–153 iterate through the list of `TreeModelListeners`, sending the `TreeModelEvent` to each. Methods `addTreeModelListener` (lines 157–161) and `removeTreeModelListener` (lines 164–168) allow `TreeModelListeners` to register and unregister for `TreeModelEvents`.

Inner-class `TreeFile` (lines 172–186) overrides method `toString` of superclass `File`. Method `toString` of class `File` returns a `String` containing the `File`’s full path name (e.g., `D:\Temp\README.TXT`). Method `toString` of class `TreeFile` (lines 182–185) overrides this method to return only the `File`’s name (e.g.,
README.TXT). Class JTree uses a DefaultTreeCellRenderer to display each node in its TreeModel. The DefaultTreeCellRenderer invokes the node’s toString method to get the text for the DefaultTreeCellRenderer’s label. Class TreeFile overrides method toString of class File so the DefaultTreeCellRenderer will show only the File’s name in the JTree, instead of the full path.

FileTreeFrame (Fig. 3.19) uses a JTree and a FileSystemModel to allow the user to view and modify a file system. The user interface consists of a JTree that shows the file system and a JTextArea that shows information about the currently selected file. Lines 33–34 create the uneditable JTextArea for displaying file information. Lines 37–38 create a FileSystemModel whose root is directory. Line 41 creates a JTree for the FileSystemModel. Line 44 sets the JTree’s editable property to true, to allow users to rename files displayed in the JTree.

```java
1 // FileTreeFrame.java
2 // JFrame for displaying file system contents in a JTree
3 // using a custom TreeModel.
4 package com.deitel.advjhtp1.mvc.tree.filesystem;
5 
6 // Java core packages
7 import java.io.*;
8 import java.awt.*;
9 import java.awt.event.*;
10 
11 // Java extension packages
12 import javax.swing.*;
13 import javax.swing.tree.*;
14 import javax.swing.event.*;
15 
16 public class FileTreeFrame extends JFrame {
17    // JTree for displaying file system
18    private JTree fileTree;
19    
20    // FileSystemModel TreeModel implementation
21    private FileSystemModel fileSystemModel;
22    
23    // JTextArea for displaying selected file's details
24    private JTextArea fileDetailsTextArea;
25    
26    // FileTreeFrame constructor
27    public FileTreeFrame( String directory )
28    {
29        super( "JTree FileSystem Viewer" );
30        
31            // create JTextArea for displaying File information
32            fileDetailsTextArea = new JTextArea();
33            fileDetailsTextArea.setEditable( false );
34
35```

Fig. 3.19 FileTreeFrame application for browsing and editing a file system using JTree and FileSystemModel (part 1 of 3).
// create FileSystemModel for given directory
FileSystemModel fileSystemModel = new FileSystemModel(
    new File( directory ) );

// create JTree for FileSystemModel
fileTree = new JTree( fileSystemModel );

// make JTree editable for renaming Files
fileTree.setEditable( true );

// add a TreeSelectionListener
fileTree.addTreeSelectionListener(
    new TreeSelectionListener() {
        // display details of newly selected File when
        // selection changes
        public void valueChanged(
            TreeSelectionEvent event )
        {
            File file = ( File )
                fileTree.getLastSelectedPathComponent();

            fileDetailsTextArea.setText( getFileDetails( file ) );
        }
    }); // end addTreeSelectionListener

// put fileTree and fileDetailsTextArea in a JSplitPane
JSplitPane splitPane = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, true,
    new JScrollPane( fileTree ),
    new JScrollPane( fileDetailsTextArea ) );

getContentPane().add( splitPane );

setDefaultCloseOperation( EXIT_ON_CLOSE );
setSize( 640, 480 );
setVisible( true );

// build a String to display file details
private String getFileDetails( File file )
{
    // do not return details for null Files
    if ( file == null )
        return "";

    // put File information in a StringBuffer
    StringBuffer buffer = new StringBuffer();
    buffer.append( "Name: " + file.getName() + "\n" );
    buffer.append( "Path: " + file.getPath() + "\n" );

    return buffer.toString();
}

Fig. 3.19 FileTreeFrame application for browsing and editing a file system using JTree and FileSystemModel (part 2 of 3).
buffer.append( "Size: " + file.length() + "\n" );
return buffer.toString();
}

// execute application
public static void main( String args[] )
{
    // ensure that user provided directory name
    if ( args.length != 1 )
        System.err.println(
            "Usage: java FileTreeFrame <path>" );
    else
        new FileTreeFrame( args[ 0 ] );
}

Fig. 3.19 FileTreeFrame application for browsing and editing a file system using JTree and FileSystemModel (part 3 of 3).

Lines 47–62 create a TreeSelectionListener to listen for TreeSelectionEvents in the JTree. Lines 55–56 of method valueChanged get the selected File object from the JTree. Lines 58–59 invoke method getFileDetails to retrieve information about the selected File and to display the details in fileDetailsTextArea. Lines 65–69 create a JSplitPane to separate the JTree and JTextArea. Lines 67 and 68 place the JTree and JTextArea in JScrollPane. Line 70 adds the JSplitPane to the JFrame.

Method getFileDetails (lines 78–91) takes a File argument and returns a String containing the File’s name, path and length. If the File argument is null, line 81 returns an empty String. Line 85 creates a StringBuffer, and lines 86–88 append
the File’s name, path and length. Line 90 invokes method **toString** of class **String-Buffer** and returns the result to the caller.

Method **main** (lines 94–104) executes the **FileTreeFrame** application. Lines 97–99 check the command-line arguments to ensure that the user provided a path for the **FileTreeModel**’s root. If the user did not provide a command-line argument, lines 98–99 display the program’s usage instructions. Otherwise, line 103 creates a new **FileTreeFrame** and passes the command-line argument to the constructor.

In this chapter, we introduced the model-view-controller architecture, the Observer design pattern and the delegate-model architecture used by several Swing components. In later chapters, we use MVC to build a Java2D paint program (Chapter 6), database-aware programs (Chapter 8, JDBC) and an Enterprise Java case study (Chapters 16–19).

**SUMMARY**

- The model-view-controller (MVC) architecture separates application data (contained in the model) from graphical presentation components (the view) and input-processing logic (the controller).
- The Java Foundation Classes (more commonly referred to as Swing components) implement a variation of MVC that combines the view and the controller into a single object, called a delegate. The delegate provides both a graphical presentation of the model and an interface for modifying the model.
- Every **JButton** has an associated **ButtonModel** for which the **JButton** is a delegate. The **ButtonModel** maintains the state information, such as whether the **JButton** is clicked, whether the **JButton** is enabled as well as a list of **ActionListeners**. The **JButton** provides a graphical presentation (e.g., a rectangle on the screen, with a label and a border) and modifies the **ButtonModel**’s state (e.g., when the user clicks the **JButton**).
- The **Observer design pattern** is a more general application of MVC that provides loose coupling between an object and its dependent objects.
- Class **java.util.Observable** represents a model in MVC, or the subject in the Observer design pattern. Class **Observable** provides method **addObserver**, which takes a **java.util.Observer** argument.
- Interface **Observer** represents the view in MVC, or the observer in the Observer design pattern. When the **Observable** object changes, it notifies each registered **Observer** of the change.
- The model-view-controller architecture requires the model to notify its views when the model changes. Method **setChanged** of class **Observable** sets the model’s **changed** flag. Method **notifyObservers** of class **Observable** notifies all **Observers** (i.e., views) of the change.
- An **Observable** object must invoke method **setChanged** before invoking method **notifyObservers**. Method **notifyObservers** invokes method **update** of interface **Observer** for each registered **Observer**.
- **JList** is a Swing component that implements the delegate-model architecture. **JList** acts as a delegate for an underlying **ListModel**.
- Interface **ListModel** defines methods for getting list elements, getting the size of the list and registering and unregistering **ListDataListeners**. A **ListModel** notifies each registered **ListDataListener** of each change in the **ListModel**.
- **JTable** is another Swing component that implements the delegate-model architecture. **JTables** are delegates for tabular data stored in **TableModel** implementations.
- **JTree** is one of the more complex Swing components that implements the delegate-model architecture. **TreeModels** represent hierarchical data, such as family trees, file systems, company
management structures and document outlines. JTreees act as delegates (i.e., combined view and controller) for TreeModels.

- To describe tree data structures, it is common to use family-tree terminology. A tree data structure consists of a set of nodes (i.e., members or elements of the tree) that are related as parents, children, siblings, ancestors and descendents.

- Interface TreeModel defines methods that describe a tree data structure suitable for representation in a JTree. Objects of any class can represent nodes in a TreeModel. For example, a Person class could represent a node in a family tree TreeModel.

- Class DefaultTreeModel provides a default TreeModel implementation. Interface TreeNode defines common operations for nodes in a DefaultTreeModel, such as get-Parent and getAllowChildren.

- Interface MutableTreeNode extends interface TreeNode to represent a node that can change, either by addition or removal of child nodes or by change of the Object associated with the node. Class DefaultMutableTreeNode provides a MutableTreeNode implementation suitable for use in a DefaultTreeModel.

- Interface TreeCellRenderer represents an object that creates a view for each node in the JTree. Class DefaultTreeCellRenderer implements interface TreeCellRenderer and extends class JLabel to provide a TreeCellRenderer default implementation.

- Interface TreeCellEditor represents an object for controlling (i.e., editing) each node in the JTree. Class DefaultTreeCellEditor implements interface TreeCellEditor and uses a JTextField to provide a TreeCellEditor default implementation.

- If the DefaultTreeModel implementation is not sufficient for an application, developers can also provide custom implementations of interface TreeModel.

**TERMINOLOGY**

ancestor
child
controller
DefaultListModel class
DefaultMutableTreeNode class
DefaultTableModel class
DefaultTreeCellEditor class
DefaultTreeCellRenderer class
DefaultTreeModel class
delegate
delegate-model architecture
descendent
getChild method of interface TreeModel
getChildAtIndex method of interface TreeModel
getChildCount method of interface TreeModel
getIndexOfChild method of interface TreeModel
isLeaf method of interface TreeModel
JList class
JTable class
JTree class
ListModel interface
ListSelectionModel interface
model
model-view-controller architecture
MutableTreeNode interface
notifyObservers method of class Observable
Observable class
Observer design pattern
Observer interface
parent
setChanged method of class Observable
sibling
TableModel interface
TreeCellEditor interface
TreeCellRenderer interface
TreeModel interface
TreeNode interface
update method of interface Observer
valueForPathChanged method of interface TreeModel
view
SELF-REVIEW EXERCISES

3.1 What more general design pattern does the model-view-controller (MVC) architecture use?
3.2 How does the variation of MVC implemented in the Swing packages differ from regular MVC?
3.3 List the Swing classes that use MVC.
3.4 What type of data does a `TableModel` contain, and what Swing class is a `TableModel` delegate?
3.5 What interfaces does a `JTree` employ to provide its delegate functionality for a `TreeModel`?

ANSWERS TO SELF-REVIEW EXERCISES

3.1 The model-view-controller architecture uses the more general Observer design pattern to separate a model (i.e., a subject) from its views (i.e., its observers).
3.2 The Swing packages use a version of MVC known as the delegate-model architecture, in which the view and controller are combined into a single object to form a delegate.
3.3 Most Swing classes use MVC, including `JButton`, `JList`, `JTable` and `JTree`.
3.4 A `TableModel` contains tabular data, such as data from a database table or spreadsheet. `JTable` is a delegate for `TableModel`s.
3.5 A `JTree` uses a `TreeCellRenderer` to provide a view of its nodes and a `TreeCellEditor` to provide a controller for its nodes.

EXERCISES

3.1 Create class `LiabilityPieChartView` as a subclass of class `AssetPieChartView` (Fig. 3.8) that includes only liability `Accounts` (i.e., `Accounts` with negative balances). Modify class `AccountManager` (Fig. 3.10) to include a `LiabilityPieChartView`, in addition to the `AssetPieChartView`.
3.2 Create a new version of class `AccountBarGraphView` (Fig. 3.7) that shows multiple `Accounts` in a single bar graph. [Hint: Try modeling your class after `AssetPieChartView` to include multiple `Accounts`]
3.3 Enhance your solution to Exercise 3.2 to allow transfers between accounts. Modify class `AccountController` (Fig. 3.9) to include a `JComboBox` to select the destination account and a `JButton` to perform the transfer.
3.4 Create a `TreeModel` implementation named `XMLTreeModel` that provides a read-only model of an XML document. Create a program that uses a `JTree` to display the XML document. If you are not familiar with XML, please see Appendices A–D.
Objectives

• To be able to use the Java 2D API to draw various shapes and general paths.
• To be able to specify Paint and Stroke characteristics of shapes displayed with Graphics2D.
• To be able to manipulate images using Java 2D image processing.
• To use the Java 3D API and Java 3D Utility classes to create three-dimensional graphics scenes.
• To manipulate the texture and lighting of three-dimensional objects with Java 3D.

Sit in reverie and watch the changing color of the waves that break upon the idle seashore of the mind.
Henry Wadsworth Longfellow

Art is not a mirror to reflect the world, but a hammer with which to shape it.
Vladimir Mayakovsky

... work transforms talent into genius.
Anna Povlova

A work that aspires, however humbly, to the condition of art should carry its justification in every line.
Joseph Conrad
4.1 Introduction

Over the past few years, developers have strived to integrate cutting-edge graphics and animation in their applets and applications. However, the original Java AWT graphics packages have provided a limited means to achieve such goals. Now, with the Java 2D™ API and Java 3D™ API, developers can implement more sophisticated graphics applications—such as games, screen savers, splash screens and 3D GUI’s.

This chapter overviews several of Java’s 2D and 3D graphics capabilities. We begin with a brief introduction to fundamental graphics topics, such as coordinate systems and graphics contexts. Next, we discuss several Java 2D capabilities, such as controlling how to fill shapes with colors and patterns. We also introduce how to blur, invert, sharpen and change the color of an image using Java 2D’s image processing capabilities. In the second half of our graphics discussion, we present the Java 3D API. Using the Java 3D utility classes, we build an application that allows the user to manipulate (rotate, scale and translate) 3D objects with a mouse. The application has a control panel that allows the user both to apply textures to 3D objects using texture mapping and to vary the lighting effects on 3D objects by changing the color of a light source.

4.2 Coordinates, Graphics Contexts and Graphics Objects

Java’s 2D coordinate system (Fig. 4.1) is a scheme for identifying every point on the screen. By default, the upper left corner of a GUI component has the coordinates (0, 0). The y-coordinate is the vertical distance moving down from the upper left corner. The x-coordinate is the horizontal distance moving right from the upper left corner.

A Java graphics context enables drawing on the screen. A Graphics object manages a graphics context by controlling how information is drawn. Graphics objects contain methods for drawing, font manipulation, color manipulation and the like. Every application that performs drawing on the screen uses Graphics object to manage the application’s graphics context.
Class `Graphics` is an abstract class (i.e., a `Graphics` object cannot be instantiated). This contributes to Java’s portability. Drawing is performed differently on each platform that supports Java so there cannot be one class that implements drawing capabilities on all systems. For example, the graphics capabilities that enable a PC running Microsoft Windows to draw a rectangle are different from the graphics capabilities that enable a UNIX workstation to draw a rectangle—and those are both different from the graphics capabilities that enable a Macintosh to draw a rectangle. For each platform, a `Graphics` subclass implements all the drawing capabilities. This implementation is hidden by the `Graphics` class, which supplies the interface that enables us to write programs that use graphics in a platform-independent manner.

Class `Component` is the superclass for many of the classes in the `java.awt` package. Method `paint` of class `Component` is called when the contents of the `Component` should be painted—either in response to the `Component` first being shown or damage needing repair—such as resizing the `Component` window. Method `paint` takes a `Graphics` reference as an argument. When a `Component` needs to be painted, the system passes a `Graphics` reference to method `paint`. This `Graphics` reference is a reference to the platform-specific `Graphics` subclass. The developer should not call method `paint` directly, because drawing graphics is an event driven process. To request the system to call `paint`, a developer can invoke method `repaint` of class `Component`. Method `repaint` requests a call to method `update` of class `Component` as soon as possible, to clear the `Component`'s background of any previous drawing. Method `update` then calls `paint` directly.

Class `JComponent`—a `Component` subclass—is the superclass for many of the classes in the `javax.swing` package. The Swing painting mechanism calls method `paintComponent` of class `JComponent` when the contents of the `JComponent` should be painted. Method `paintComponent`—which takes as an argument a `Graphics` object—helps the Swing components paint properly. The `Graphics` object is passed to the `paintComponent` method by the system when a `paintComponent` operation is required for a `JComponent`. The developer should not call method `paintComponent` directly. If the developer needs to call `paintComponent`, a call is made to method `repaint` of class `Component`—exactly as discussed earlier for method `repaint` of class `Component`.

![Java coordinate system. Units are measured in pixels.](image)
4.3 Java 2D API

The *Java 2D™ API* provides advanced 2D graphics capabilities for developers who require detailed and complex graphical manipulations in their programs. The Java 2D API is part of the Java 2 Platform, Standard Edition. The Java 2D API includes features for processing line art, text and images in packages `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` and `java.awt.image.renderable`. Figure 4.2 describes several of the Java 2D classes and interfaces covered in this chapter.

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classes and interfaces from package java.awt</strong></td>
<td></td>
</tr>
<tr>
<td>Graphics2D</td>
<td>Graphics subclass for rendering 2D shapes, text and images.</td>
</tr>
<tr>
<td>BasicStroke</td>
<td>Defines a basic set of rendering attributes for the outlines of graphics primitives.</td>
</tr>
<tr>
<td>GradientPaint</td>
<td>Provides a way to fill and outline 2D shapes with a linear color gradient.</td>
</tr>
<tr>
<td>TexturePaint</td>
<td>Provides a way to fill and outline shapes with texture images.</td>
</tr>
<tr>
<td>Paint</td>
<td>Defines how color patterns can be generated for rendering operations.</td>
</tr>
<tr>
<td>Shape</td>
<td>Provides definitions for geometrical objects.</td>
</tr>
<tr>
<td>Stroke</td>
<td>Provides methods for obtaining the outline of a geometrical shape.</td>
</tr>
<tr>
<td><strong>Classes and interfaces from package java.awt.geom</strong></td>
<td></td>
</tr>
<tr>
<td>GeneralPath</td>
<td>Represents a path constructed from straight lines, quadratic curves and cubic curves.</td>
</tr>
<tr>
<td>Line2D</td>
<td>Represents a line in coordinate space.</td>
</tr>
<tr>
<td>RectangularShape</td>
<td>Base class for geometrical shapes with rectangular frames. Subclasses include Arc2D, Ellipse2D, Rectangle2D and RoundRectangle2D.</td>
</tr>
<tr>
<td>BufferedImage</td>
<td>Describes an Image with a buffer of colored pixel data composed of a ColorModel and a Raster.</td>
</tr>
<tr>
<td>ColorModel</td>
<td>Defines methods for translating a numerical pixel value to a color.</td>
</tr>
<tr>
<td><strong>Classes and interfaces from package java.awt.image</strong></td>
<td></td>
</tr>
<tr>
<td>Raster</td>
<td>Is part of a BufferedImage that describes sample values in a rectangular array of pixels.</td>
</tr>
<tr>
<td>Kernel</td>
<td>Describes a 2D array used for filtering BufferedImage.</td>
</tr>
</tbody>
</table>

*Fig. 4.2 Some Java 2D classes and interfaces (part 1 of 2).*
Class `java.awt.Graphics2D` enables drawing with the Java 2D API. Class `Graphics2D` is a subclass of class `Graphics`, so it has all the capabilities for managing the application’s graphics context discussed earlier in this chapter. To access the `Graphics2D` capabilities, we cast the `Graphics` reference passed to `paint` to a `Graphics2D` reference.

Java 2D can render three types of built-in graphics objects—termed `graphics primitives`—images, text and geometrical shapes. There are seven `Graphics2D` state attributes that determine how graphics primitives are rendered—clipping, compositing, font, paint, rendering hints, stroke and transforms. Figure 4.3 describes each of these seven attributes. The attributes form a pipeline that processes the graphics primitives to produce the final image. The first stage in the pipeline determines which of the primitives to render. A `draw method` then draws the primitive—method `draw` for shapes, method `drawString` for text and method `drawImage` for images. The pipeline applies any transformations, fills and strokes during the drawing process. The next stage is to rasterize the drawn shape—convert the shape to a two-dimensional array of numerical pixel values called a `raster`. At this stage, the pipeline invokes any image-processing operations on the raster. The raster is then clipped, colored and combined with the current drawing—known as compositing. Finally, the image is rendered—drawn—on an output device, such as a screen or printer.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipping</td>
<td>Defines the area in which rendering operations take effect. Any geometrical shape, including text, can be used as a clipping region.</td>
</tr>
<tr>
<td>Compositing</td>
<td>Is a Set of blending rules that control how the pixels in a source image mix with the pixels in a destination image.</td>
</tr>
<tr>
<td>Font</td>
<td>Fonts are created from shapes that represent the characters to be drawn—called glyphs. Text is rendered by drawing and filling the glyphs.</td>
</tr>
<tr>
<td>Paint</td>
<td>Determines the colors, patterns and gradients for filling and outlining a shape.</td>
</tr>
<tr>
<td>Rendering Hints</td>
<td>Specify techniques and methods that help to optimize drawing.</td>
</tr>
</tbody>
</table>

Class `java.awt.Graphics2D` enables drawing with the Java 2D API. Class `Graphics2D` is a subclass of class `Graphics`, so it has all the capabilities for managing the application’s graphics context discussed earlier in this chapter. To access the `Graphics2D` capabilities, we cast the `Graphics` reference passed to `paint` to a `Graphics2D` reference.

Java 2D can render three types of built-in graphics objects—termed `graphics primitives`—images, text and geometrical shapes. There are seven `Graphics2D` state attributes that determine how graphics primitives are rendered—clipping, compositing, font, paint, rendering hints, stroke and transforms. Figure 4.3 describes each of these seven attributes. The attributes form a pipeline that processes the graphics primitives to produce the final image. The first stage in the pipeline determines which of the primitives to render. A `draw method` then draws the primitive—method `draw` for shapes, method `drawString` for text and method `drawImage` for images. The pipeline applies any transformations, fills and strokes during the drawing process. The next stage is to rasterize the drawn shape—convert the shape to a two-dimensional array of numerical pixel values called a `raster`. At this stage, the pipeline invokes any image-processing operations on the raster. The raster is then clipped, colored and combined with the current drawing—known as compositing. Finally, the image is rendered—drawn—on an output device, such as a screen or printer.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clipping</td>
<td>Defines the area in which rendering operations take effect. Any geometrical shape, including text, can be used as a clipping region.</td>
</tr>
<tr>
<td>Compositing</td>
<td>Is a Set of blending rules that control how the pixels in a source image mix with the pixels in a destination image.</td>
</tr>
<tr>
<td>Font</td>
<td>Fonts are created from shapes that represent the characters to be drawn—called glyphs. Text is rendered by drawing and filling the glyphs.</td>
</tr>
<tr>
<td>Paint</td>
<td>Determines the colors, patterns and gradients for filling and outlining a shape.</td>
</tr>
<tr>
<td>Rendering Hints</td>
<td>Specify techniques and methods that help to optimize drawing.</td>
</tr>
</tbody>
</table>
The Java 2D API provides hints and rules that instruct the graphics engine how to perform these operations. The following sections present several features of image and geometrical shape-rendering processes.

### 4.3.1 Java 2D Shapes

In this section, we present several Java 2D shape classes from package `java.awt.geom`, including `Ellipse2D.Double`, `Line2D.Double`, `Rectangle2D.Double`, `RoundRectangle2D.Double` and `Arc2D.Double`. Each class represents a shape with dimensions specified as double-precision floating-point values. Each class can also be represented with single-precision floating-point values (e.g., `Ellipse2D.Float`). In each case, class `Double` is a static inner class contained in the class to the left of the dot operator (e.g., `Ellipse2D`).

Class `Shapes` (Fig. 4.4) demonstrates several Java 2D shapes and rendering attributes (such as thick lines), filling shapes with patterns and drawing dashed lines. These are just a few of the many capabilities Java 2D provides.

```java
// Shapes.java
// Shapes demonstrates some Java 2D shapes.

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;

// Java extension packages
import javax.swing.*;

public class Shapes extends JFrame {
    // constructor method
    public Shapes() {
        super( "Drawing 2D shapes" );
    }
}
```

**Fig. 4.3** The seven state attributes of a Java 2D graphics context (part 2 of 2).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stroke</td>
<td>Determines the outline of the shape to be drawn.</td>
</tr>
<tr>
<td>Transform</td>
<td>Defines ways to perform linear transformations—an operation that changes the shape of an image.</td>
</tr>
</tbody>
</table>
// draw shapes using Java 2D API
public void paint( Graphics g )
{
  // call superclass' paint method
  super.paint( g );

  // get Graphics 2D by casting g to Graphics2D
  Graphics2D graphics2D = ( Graphics2D ) g;

  // draw 2D ellipse filled with blue-yellow gradient
  graphics2D.setPaint( new GradientPaint( 5, 30, Color.blue, 35, 100, Color.yellow, true ) );
  graphics2D.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );

  // draw 2D rectangle in red
  graphics2D.setPaint( Color.red );
  graphics2D.setStroke( new BasicStroke( 10.0f ) );
  graphics2D.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );

  // draw 2D rounded rectangle with BufferedImage background
  BufferedImage bufferedImage = new BufferedImage( 10, 10, BufferedImage.TYPE_INT_RGB );
  Graphics2D graphics = bufferedImage.createGraphics();
  graphics.setColor( Color.yellow ); // draw in yellow
  graphics.fillRect( 0, 0, 10, 10 ); // draw filled rectangle
  graphics.setColor( Color.black ); // draw in black
  graphics.drawRect( 1, 1, 6, 6 ); // draw rectangle
  graphics.setColor( Color.blue ); // draw in blue
  graphics.fillRect( 1, 1, 3, 3 ); // draw filled rectangle
  graphics.setColor( Color.red ); // draw in red
  graphics.fillRect( 4, 4, 3, 3 ); // draw filled rectangle

  // paint buffImage into graphics context of JFrame
  graphics2D.setPaint( new TexturePaint( bufferedImage, new Rectangle( 10, 10 ) ) );
  graphics2D.fill( new RoundRectangle2D.Double( 155, 30, 75, 100, 50, 50 ) );

  // draw 2D pie-shaped arc in white
  graphics2D.setPaint( Color.white );
  graphics2D.setStroke( new BasicStroke( 6.0f ) );
  graphics2D.draw( new Arc2D.Double( 240, 30, 75, 100, 0, 270, Arc2D.PIE ) );

  // draw 2D lines in green and yellow
  graphics2D.setPaint( Color.green );
  graphics2D.draw( new Line2D.Double( 395, 30, 320, 150 ) );

  float dashes[] = { 10, 2 };
  graphics2D.setPaint( Color.yellow );
Line 28 casts the `Graphics` reference received by `paint` to a `Graphics2D` reference to allow access to Java 2D features. The first shape we draw is an oval filled with gradually changing colors. Lines 31–32 invoke method `setPaint` of class `Graphics2D` to set the `Paint` object that determines the color for the shape to display. A `Paint` object is an object of any class that implements interface `java.awt.Paint`. The `Paint` object can be something as simple as one of the predefined `Color` objects (class `Color` implements `Paint`), or the `Paint` object can be an instance of the Java 2D API’s `GradientPaint`, `SystemColor` or `TexturePaint` classes. In this case, we use a `GradientPaint` object.

Class `GradientPaint` paints a shape in gradually changing colors—a `gradient`. The `GradientPaint` constructor used here requires seven arguments. The first two arguments specify the starting coordinate for the gradient. The third argument specifies the starting `Color` for the gradient. The fourth and fifth arguments specify the ending coordinate for the gradient. The sixth argument specifies the ending `Color` for the gradient. The last argument specifies whether the gradient is cyclic (`true`) or acyclic (`false`). The two coordinates determine the direction of the gradient. The second coordinate (35, 100) is down and to the right of the first coordinate (5, 30); therefore, the gradient goes down and to the right at an angle. Since this gradient is cyclic (`true`), the color starts with blue, gradually becomes yellow, then gradually returns to blue. If the gradient is acyclic, the color transitions from the first color specified (e.g., blue) to the second color (e.g., yellow).
Line 33 uses method `fill` of class `Graphics2D` to draw a filled `Shape` object. The `Shape` object is an instance of any class that implements interface `Shape` (package `java.awt`)—in this case, an instance of class `Ellipse2D.Double`. The `Ellipse2D.Double` constructor receives four arguments that specify the bounding rectangle for the ellipse to display.

Next we draw a red rectangle with a thick border. Line 36 uses method `setPaint` to set the `Paint` object to `Color.red`. Line 37 uses method `setStroke` of class `Graphics2D` to set the characteristics of the rectangle’s border. Method `setStroke` requires a `Stroke` object as its argument. The `Stroke` object is an instance of any class that implements interface `Stroke` (package `java.awt`)—in this case, an instance of class `BasicStroke`. Class `BasicStroke` provides a variety of constructors to specify the line width, how the line ends (called the `end caps`), how lines join together (called `line joins`) and the dash attributes of the line (if it is a dashed line). The constructor here specifies that the line should be 10 pixels wide.

Lines 38–39 invoke method `draw` of `Graphics2D` to draw a `Shape` object—in this case, an instance of class `Rectangle2D.Double`. The `Rectangle2D.Double` constructor receives four arguments specifying the upper left x-coordinate, upper left y-coordinate, width and height of the rectangle measured in pixels.

Next we draw a rounded rectangle filled with a pattern created in a `BufferedImage` (package `java.awt.image`) object. Lines 42–43 create the `BufferedImage` object. Class `BufferedImage` can produce images in color and gray scale. This particular `BufferedImage` is 10 pixels wide and 10 pixels tall. The third constructor argument `BufferedImage.TYPE_INT_RGB` specifies that the image is stored in color using the Red Green Blue (RGB) color scheme.

To create the fill pattern for the rounded rectangle, we must first draw into the `BufferedImage`. Line 45 creates a `Graphics2D` object for drawing on the `BufferedImage`. Lines 46–53 use methods `setColor`, `fillRect` and `drawRect` (discussed earlier in this chapter) to create the pattern.

Lines 56–57 set the `Paint` object to a new `TexturePaint` (package `java.awt`) object. A `TexturePaint` object uses the image stored in its associated `BufferedImage` as the fill texture for a filled-in shape. The second argument specifies the `Rectangle` area from the `BufferedImage` that will be replicated through the texture. In this case, the `Rectangle` is the same size as the `BufferedImage`. However, a smaller portion of the `BufferedImage` can be used.

Lines 58–59 invoke method `fill` of `Graphics2D` to draw a filled `Shape` object—`RoundRectangle2D.Double`. The `RoundRectangle2D.Double` constructor receives six arguments specifying the rectangle dimensions and the arc width and arc height—measured in pixels—used to determine the rounding of the corners.

Next we draw a oblong arc with a thick white line. Line 62 sets the `Paint` object to `Color.white`. Line 63 sets the `Stroke` object to a new `BasicStroke` for a line 6 pixels wide. Lines 64–65 use method `draw` of class `Graphics2D` to draw a `Shape` object—in this case, an `Arc2D.Double`. The `Arc2D.Double` constructor’s first four arguments specifying the upper left x-coordinate, upper left y-coordinate, width and height of the bounding rectangle for the arc. The fifth argument specifies the start angle measured in degrees. The sixth argument specifies the arc angle. The start angle and arc angles are measured relative to the shape’s bounding rectangle. The last argument specifies how the
arc is closed. Constant \texttt{Arc2D.PIE} indicates that the arc is closed by drawing two lines. One line from the arc’s starting point to the center of the bounding rectangle and one line from the center of the bounding rectangle to the ending point. Class \texttt{Arc2D} provides two other \texttt{static} constants for specifying how the arc is closed. Constant \texttt{Arc2D.CHORD} draws a line from the starting point to the ending point. Constant \texttt{Arc2D.OPEN} specifies that the arc is not closed.

Finally, we draw two lines using \texttt{Line2D} objects—one solid and one dashed. Line 68 sets the \texttt{Paint} object to \texttt{Color.green}. Line 69 uses method \texttt{draw} of class \texttt{Graphics2D} to draw a \texttt{Shape} object—in this case, an instance of class \texttt{Line2D.Double}. The \texttt{Line2D.Double} constructor’s arguments specify starting coordinates and ending coordinates of the line.

Line 71 defines a two-element \texttt{float} array. This array describes the length—in pixels—of the dashes and spaces in the dashed line. In this case, each dash will be 10 pixels long and each space will be two pixels long. To create dashes of different lengths in a pattern, simply provide the lengths of each dash as an element in the array. Line 73 sets the \texttt{Paint} object to \texttt{Color.yellow}. Lines 74–76 set the \texttt{Stroke} object to a new \texttt{BasicStroke}. The line will be 4 pixels wide and will have rounded ends (\texttt{BasicStroke.CAP_ROUND}). If lines join together (as in a rectangle at the corners), the joining of the lines will be rounded (\texttt{BasicStroke.JOIN_ROUND}). The \texttt{dashes} argument specifies the dash lengths for the line. The last argument indicates the starting subscript in the \texttt{dashes} array for the first dash in the pattern. Line 77 then draws a line with the current \texttt{Stroke}.

Next we present a general path—a shape constructed from lines and complex curves. A general path is represented with an object of class \texttt{GeneralPath} (package \texttt{java.awt.geom}). Class \texttt{Shapes2} (Fig. 4.5) demonstrates drawing a general path in the shape of a five-pointed star.
// draw general paths
public void paint( Graphics g )
{
    // call superclass' paint method
    super.paint( g );

    int xPoints[] =
    { 55, 67, 109, 73, 83, 27, 37, 1, 43 };  
    int yPoints[] =
    { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };

    Graphics2D graphics2D = ( Graphics2D ) g;

    // create a star from a series of points
    GeneralPath star = new GeneralPath();

    // set the initial coordinate of the General Path
    star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );

    // create the star--this does not draw the star
    for ( int count = 1; count < xPoints.length; count++ )
        star.lineTo( xPoints[ count ], yPoints[ count ] );

    // close the shape
    star.closePath();

    // translate the origin to (200, 200)
    graphics2D.translate( 200, 200 );

    // rotate around origin and draw stars in random colors
    for ( int count = 1; count <= 20; count++ ) {
        // rotate coordinate system
        graphics2D.rotate( Math.PI / 10.0 );

        // set random drawing color
        graphics2D.setColor( new Color(  
            ( int ) ( Math.random() * 256 ),  
            ( int ) ( Math.random() * 256 ),  
            ( int ) ( Math.random() * 256 ) ) );

        // draw filled star
        graphics2D.fill( star );
    }

} // end method paint

// execute application
public static void main( String args[] )
{
    Shapes2 application = new Shapes2();
    application.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE );

Fig. 4.5 Demonstrating Java 2D paths (part 2 of 3).
Lines 29–32 define two `int` arrays representing the x- and y-coordinates of the points in the star. Line 37 defines `GeneralPath` object `star`. Line 40 uses method `moveTo` of class `GeneralPath` to specify the first point in the `star`. The `for` structure at lines 43–44 uses method `lineTo` of class `GeneralPath` to draw a line to the next point in the `star`. Each new call to `lineTo` draws a line from the previous point to the current point. Line 47 uses method `closePath` of class `GeneralPath` to draw a line from the last point to the point specified in the last call to `moveTo`. This completes the general path.

Line 50 uses method `translate` of class `Graphics2D` to move the drawing origin to location (200, 200). All drawing operations now use location (200, 200) as (0, 0). The `for` structure at lines 53–65 draws the `star` 20 times by rotating it around the new origin point. Line 56 uses method `rotate` of class `Graphics2D` to rotate the next displayed shape. The argument specifies the rotation angle in radians (360° = 2π radians). Line 65 uses `Graphics2D` method `fill` to draw a filled version of the `star`.

### 4.3.2 Java 2D Image Processing

`Image processing` is the manipulation of digital images by applying `filters`—mathematical operations that change images. Java 2D provides an image-processing API to shield developers from the mathematics behind filters. `Compression filters`, `measurement filters` and `enhancement filters` constitute the three major image-processing categories. Compression filters reduce a digital image’s memory usage, resulting in reduced storage size and faster transmission of complex digital images. Some common applications of compression filters include high-definition television (HDTV), video phones and virtual reality. Measurement
filters collect data from digital images. Measurement filters play a crucial role in the field of image recognition and machine vision (e.g., for printed circuit board inspection and assembly-line welding robots). Enhancement filters—filters that alter certain physical aspects of an image—often restore corrupted images to their original form. Sometimes, the processes of creating, storing or transmitting a digital image introduces data corruption such as noise, motion blurring and color loss. Enhancement filters can remove noise, sharpen edges and brighten colors to recover the original image. For example, satellite images use enhancement filters to remove noise created from capturing images at such lengthy distances.

Java 2D image-processing filters operate on objects of class BufferedImage, which separates image data into two components—a Raster and a ColorModel. A Raster—composed of a DataBuffer and a SampleModel—organizes and stores the data that determine a pixel’s color. Each pixel is composed of samples—numbers that represent the pixel’s color components. The DataBuffer stores the raw sample data for an image. The SampleModel accesses the sample values in the DataBuffer for any given pixel. The ColorModel is an interpreter for the Raster, taking the sample values for each pixel in the Raster and converting them to the appropriate color. The ColorModel converts the sample data to different colors depending on the color scale of the image. Two common color scales are grayscale and RGB. In grayscale, every pixel is represented by one sample interpreted as a color between black and white. In RGB, each pixel is represented by three samples that correspond to the red, green and blue color components of the pixel.

This section presents an application that demonstrates how to create and filter BufferedImage. We build filters that blur, sharpen, invert and change the color scale of a BufferedImage. These are “fundamental” filters found in mass graphics programs, such as Paint Shop Pro. Our application allows the user to apply a series of filters to a BufferedImage to demonstrate the effects of multiple filters. Sample filter results appear in the screen captures of Fig. 4.13. The application consists of three distinct parts:

1. ImagePanel—a JPanel extended to provide image-processing capabilities.

2. Java2DImageFilter—an interface for image-processing filters that will alter the image in an ImagePanel. The classes that implement interface Java2DImageFilter include BlurFilter, SharpenFilter, InvertFilter and ColorChangeFilter.

3. Java2DExample—a GUI that displays the filtered image and presents the user with a menu for selecting image filters.

Class ImagePanel (Fig. 4.6) allows a user to experiment with applying various filters to an image. ImagePanel contains an image and methods for filtering that image. Lines 18–19 declare two BufferedImage references—displayImage and originalImage. The image filters manipulate displayImage, and originalImage stores a copy of the original image so the user can view the original image.

```java
1 // ImagePanel.java
2 // ImagePanel contains an image for display. The image is
3 // converted to a BufferedImage for filtering purposes.
4 package com.deitel.advjhtp1.java2d;
```

Fig. 4.6 Class ImagePanel allows for displaying and filtering BufferedImage (part 1 of 3).
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.net.*;

import javax.swing.*;
import javax.swing.event.*;

public class ImagePanel extends JPanel {
    private BufferedImage displayImage; // filtered image
    private BufferedImage originalImage; // original image
    private Image image; // image to load

    // ImagePanel constructor
    public ImagePanel( URL imageURL ) {
        image = Toolkit.getDefaultToolkit().createImage( imageURL );

        MediaTracker mediaTracker = new MediaTracker( this );
        mediaTracker.addImage( image, 0 );

        try {
            mediaTracker.waitForAll();
        } catch ( InterruptedException interruptedException ) {
            interruptedException.printStackTrace();
        }

        originalImage = new BufferedImage( image.getWidth( null ),
                                            image.getHeight( null ), BufferedImage.TYPE_INT_RGB );
        displayImage = originalImage;

        Graphics2D graphics = displayImage.createGraphics();
        graphics.drawImage( image, null, null );
    }

    // apply Java2DImageFilter to Image
    public void applyFilter( Java2DImageFilter filter ) {
        // Fig. 4.6 Class ImagePanel allows for displaying and filtering BufferedImage (part 2 of 3).
The `ImagePanel` constructor (lines 23–52) accepts as an argument a URL that specifies the file containing the image to filter. Lines 25–26 create an `Image` object—`image`—from this file. Lines 29–30 instantiate a `MediaTracker` for `image` loading. Method `waitForAll` (line 34) of class `MediaTracker` ensures that `image` is loaded into memory before we filter this image.

Lines 43–46 create `BufferedImages displayImage` and `originalImage`. The `BufferedImage` constructor accepts three arguments—the image’s width, height and type. We use predefined type `TYPE_INT_RGB`, which defines three 8-bit segments each representing a red, green and blue color components. Line 49 creates a `Graphics2D` object for rendering `displayImage`. Line 50 renders the loaded `image` on `ImagePanel` using method `drawImage` of class `Graphics2D`.
Method **applyFilter** (lines 55–60) applies a **Java2DImageFilter** to **displayImage**. Line 58 invokes method **processImage** of class **Java2DImageFilter**, which passes **displayImage** as a parameter. Method **processImage** applies an image filter to **displayImage**. Line 59 calls method **repaint**, which indicates that **ImagePanel** needs to be redrawn. In turn, a system call is made to method **paintComponent** of class **ImagePanel**. Line 77 casts the **Graphics** object to a **Graphics2D** object to access **Graphics2D** methods. The **Graphics2D**’s method **drawImage** (line 78) renders **displayImage** in the **ImagePanel**.

We provide a means to reconstruct the original image after the program applies filters to **displayImage**. Method **displayOriginal** (lines 63–71) creates a new **BufferedImage** that contains a copy of **originalImage** so the user can apply a new set of filters to **displayImage**. Lines 65–66 recreate **displayImage** as a new **BufferedImage**. Line 68 creates a **Graphics2D** for **displayImage**. Line 69 calls method **drawImage** of class **Graphics2D**, which draws **originalImage** into **displayImage**.

We now implement our image-processing filters—**BlurFilter**, **SharpenFilter**, **InvertFilter** and **ColorFilter**. Our filters implement interface **Java2DImageFilter** (Fig. 4.7). Classes that implement **Java2DImageFilter** must implement method **processImage** (line 13). Method **processImage** accepts a **BufferedImage** to filter and returns the filtered **BufferedImage**.

The **Java2DImageFilters** in this application use well-known Java 2D image-processing operations. Java 2D has several image filters that operate on **BufferedImages**. Interfaces **BufferedImageOp** and **RasterOp** serve as the base classes for Java 2D image filters. Method **filter** of interfaces **BufferedImageOp** and **RasterOp** takes as arguments two images—the **source** image and the **destination** image. All classes that implement **BufferedImageOp** and **RasterOp** apply a filter to the source image to produce the destination image. A **BufferedImageOp** processes a **BufferedImage**, while a **RasterOp** processes only the **Raster** associated with a **BufferedImage**. Several Java 2D image filters implement **BufferedImageOp** and/or **RasterOp** (Fig. 4.8).

```java
// Java2DImageFilter.java
// Java2DImageFilter is an interface that defines method
// processImage for applying a filter to an Image.
package com.deitel.advjhtp1.java2d;

public interface ImageFilter {
    // apply filter to Image
    public BufferedImage processImage( BufferedImage image );
}
```

**Fig. 4.7**  **Java2DImageFilter** interface for creating Java 2D image filters.
We now present each `Java2DImageFilter` in our application. Class `InvertFilter` (Fig. 4.9), which implements interface `Java2DImageFilter`, inverts the color of the pixels in a `BufferedImage`. Each pixel consists of three samples—8-bit R, G and B integers. An 8-bit color sample takes on an integer in the range 0–255. By inverting the numerical value of the pixel sample, we can invert the color of the pixel. Line 15 creates an array to hold the inverted integers. Lines 17–18 invert the array values.

InvertFilter uses a `LookupOp`—a subclass of `BufferedImageOp`—to invert the colors. Class `BufferedImageOp`—the base class for most Java 2D filters—operates on two images (the source image and the destination image). All classes that implement `BufferedImageOp` filter the source image to produce the destination image. A `LookupOp` is an array indexed by source pixel color values and contains destination pixel color values. Lines 21–22 create a new `LookupOp—invertFilter`. The `LookupOp` constructor takes as arguments a `ByteLookupTable` that contains the lookup array table—invertArray—and a `RenderingHints`. The `RenderingHints` object describes optimizations for the rendering engine. In this application, no optimizations are needed, so `RenderingHints` is `null`. Line 25 invokes method `filter` of class `LookupOp`, which processes `image` with `invertFilter` and returns the filtered image.

<table>
<thead>
<tr>
<th>Class</th>
<th>Implements Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineTransformOp</td>
<td>BufferedImageOp</td>
<td>Performs linear mapping from 2D coordinates in the source image to 2D coordinates in the destination image. (Example: Rotate an image about a point in the image.)</td>
</tr>
<tr>
<td>BandCombineOp</td>
<td>RasterOp</td>
<td>Performs a linear combination of the bands in a Raster. (Example: Change the color palette in an image.)</td>
</tr>
<tr>
<td>ColorConvertOp</td>
<td>BufferedImageOp</td>
<td>Performs color conversion on each pixel in the source image. (Example: Convert from RGB color to gray scale.)</td>
</tr>
<tr>
<td>ConvolveOp</td>
<td>BufferedImageOp</td>
<td>Combines source pixel values with surrounding pixel values to derive destination pixel values. (Example: Sharpen edges in an image.)</td>
</tr>
<tr>
<td>LookupOp</td>
<td>BufferedImageOp</td>
<td>Performs a lookup operation on the source image to create the destination image. (Example: Invert the RGB colors in an image.)</td>
</tr>
<tr>
<td>RescaleOp</td>
<td>BufferedImageOp</td>
<td>Rescale the data in the source image by a scalar plus offset. (Example: Darken the coloring of an image.)</td>
</tr>
</tbody>
</table>

Fig. 4.8 Classes that implement `BufferedImageOp` and `RasterOp`. 

We now present each `Java2DImageFilter` in our application. Class `InvertFilter` (Fig. 4.9), which implements interface `Java2DImageFilter`, inverts the color of the pixels in a `BufferedImage`. Each pixel consists of three samples—8-bit R, G and B integers. An 8-bit color sample takes on an integer in the range 0–255. By inverting the numerical value of the pixel sample, we can invert the color of the pixel. Line 15 creates an array to hold the inverted integers. Lines 17–18 invert the array values.

InvertFilter uses a `LookupOp`—a subclass of `BufferedImageOp`—to invert the colors. Class `BufferedImageOp`—the base class for most Java 2D filters—operates on two images (the source image and the destination image). All classes that implement `BufferedImageOp` filter the source image to produce the destination image. A `LookupOp` is an array indexed by source pixel color values and contains destination pixel color values. Lines 21–22 create a new `LookupOp—invertFilter`. The `LookupOp` constructor takes as arguments a `ByteLookupTable` that contains the lookup array table—invertArray—and a `RenderingHints`. The `RenderingHints` object describes optimizations for the rendering engine. In this application, no optimizations are needed, so `RenderingHints` is `null`. Line 25 invokes method `filter` of class `LookupOp`, which processes `image` with `invertFilter` and returns the filtered image.
Class **SharpenFilter** (Fig. 4.10) is a filter that detects and enhances *edges*—differences in the sample values of neighboring pixels—in an image. A sharpening filter first detects edges by determining differences in neighboring pixel sample values, then enhances the edge by increasing the difference between the sample values. **SharpenFilter** uses a **ConvolveOp**—another subclass of **BufferedImageOp**—to create the sharpening filter. A **ConvolveOp** combines the colors of a source pixel and its surrounding neighbors to determine the color of the corresponding destination pixel. Lines 15–18 create **sharpenMatrix**—the values used in the **ConvolveOp**. Lines 21–23 create the **ConvolveOp**—**sharpenFilter**—passing three parameters (a **Kernel**, an integer edge hint and a **RenderingHints** object). The **Kernel**—a 2D array—specifies how a **ConvolveOp** filter should combine neighboring pixel values. Every **ConvolveOp** is built from a **Kernel**. The **Kernel** constructor takes as arguments a width, height and an array of values. Using these arguments, a two-dimensional array is constructed from the array values. Edge hints instruct the filter how to alter pixels at the perimeter of the image. **EDGE_NO_OP** (line 23) instructs **sharpenFilter** to copy the source pixels at the perimeter of **image** directly to the destination image without modification. Line 26 invokes method **filter** of class **ConvolveOp**, which takes as an argument a **BufferedImage**. Method **filter** returns the filtered image.
Class **BlurFilter** (Fig. 4.11) uses a **ConvolveOp** to blur a **BufferedImage**. A blurring filter smooths distinct edges by averaging each pixel value with that of its eight neighboring pixels. Lines 14–17 create the **blurMatrix**—an array of values for constructing the **Kernel**. Lines 20–21 create a **ConvolveOp blurFilter** using the default constructor, which takes as an argument a **Kernel** constructed from the **blurMatrix**. The default constructor uses **EDGE_ZERO_FILL** for the edge hint and a **null RenderingHints**. **EDGE_ZERO_FILL** specifies that pixels at the outer edge of the destination **BufferedImage** be set to 0—this is the default. Line 24 invokes **blurFilter’s** method **filter** on **image**.

```java
// BlurFilter.java
// BlurFilter, which implements Java2DImageFilter, blurs
// the edges in a BufferedImage.
package com.deitel.javajhtp1.java2d;

// Java core packages
import java.awt.image.*;

public class BlurFilter implements Java2DImageFilter {
   // apply edge-blurring filter to BufferedImage
   public BufferedImage processImage( BufferedImage image )
   {
      // array used to detect edges in image
      float[] blurMatrix = {
         0.0f, -1.0f, 0.0f,
         -1.0f, 5.0f, -1.0f,
         0.0f, -1.0f, 0.0f };

      // create filter to blur edges
      BufferedImageOp blurFilter =
         new ConvolveOp( new Kernel( 3, 3, blurMatrix ),
                          ConvolveOp.EDGE_ZERO_FILL, null );

      // apply blurFilter to displayImage
      return blurFilter.filter( image, null );
   }
}
```

**Fig. 4.11** **BlurFilter** blurs the colors in a **BufferedImage** (part 1 of 2).
Class **ColorFilter** (Fig. 4.12) alters the color bands in a **BufferedImage**. There are three color bands in a **TYPE_INT_RGB BufferedImage**—red, green and blue. Each color band is defined by three coefficients that represent the R, G and B components in the band. The standard red color band consists of **1.0f R, 0.0f G and 0.0f B** color components—i.e. the standard red band consists entirely of red. Likewise, the standard green color band consists of **0.0f R, 1.0f G and 0.0f B**, while the standard blue color band consists of **0.0f R, 0.0f G and 1.0f B**. We can change image colors by altering the values of the R, G and B coefficients in a color band.

---

```java
9    // apply blurring filter to BufferedImage
10   public BufferedImage processImage( BufferedImage image )
11   {
12       // array used to blur BufferedImage
13       float[] blurMatrix = {
14           1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
15           1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
16           1.0f / 9.0f, 1.0f / 9.0f, 1.0f / 9.0f,
17       };
18
19       // create ConvolveOp for blurring BufferedImage
20       BufferedImageOp blurFilter = new ConvolveOp(
21           new Kernel( 3, 3, blurMatrix ) );
22
23       // apply blurFilter to BufferedImage
24       return blurFilter.filter( image, null );
25
26   } // end method processImage
27 }
```

**Fig. 4.11** **BlurFilter** blurs the colors in a **BufferedImage** (part 2 of 2).

```java
9    // ColorFilter.java
10   // ColorFilter is an Java2DImageFilter that alters the RGB color bands in a BufferedImage.
11   package com.deitel.advjhtp1.java2d;
12
13   import java.awt.image.*;
14
15   public class ColorFilter implements Java2DImageFilter {
16
17       // apply color-change filter to BufferedImage
18       public BufferedImage processImage( BufferedImage image )
19       {
20           // create array used to change RGB color bands
21           float[][] colorMatrix = {
22               { 1.0f, 0.0f, 0.0f },
23               { 0.5f, 1.0f, 0.5f },
24               { 0.2f, 0.4f, 0.6f } };
25
26       } // end method processImage
27 }
```

**Fig. 4.12** **ColorFilter** changes the colors in a **BufferedImage** (part 1 of 2).
Lines 15–18 create `colorMatrix`—a 2D array that represents a nonstandard color space—the aggregation of red, green and blue color bands. The red band (line 16) is the same as in the standard space. The green and blue bands (lines 17–18) assume color values from all three color components—green and blue will contain elements of R, G and B.


Line 25 calls method `getRaster` of class `BufferedImage`, which returns the `Raster` associated with `image`—sourceRaster. Lines 27–28 call method `createCompatibleWritableRaster` of class `Raster`, which returns `displayRaster`—a `WritableRaster` compatible with `sourceRaster`. Compatible `Rasters` contain the same number of bands. A `WritableRaster` allows sample data to be written while a `Raster` is read-only. Line 31 invokes method `filter` of class `BandCombineOp`, which takes as arguments a source `Raster` and a destination `WritableRaster`. The source `Raster` is filtered and written into the destination `WritableRaster`.

Lines 34–35 construct a `BufferedImage`. This `BufferedImage` constructor takes four arguments—a `ColorModel`, a `Raster`, a `boolean` and a `Hashtable`. We use the `ColorModel` of the original image, accessed through method `getColorModel` of class `Image` (line 34). Class `ColorModel` converts `Raster` data to colors depending on the color scale of the image. The `Raster` argument to the `BufferedImage` constructor is our `displayRaster`. The `boolean` value indicates whether the `Raster` has been premultiplied with alpha values. Each pixel is a small square. A curve in an image may require that only a portion of a pixel be colored—the alpha values tell the `Raster` how much of the pixel to cover. The `Hashtable` contains `String/object` properties and is `null` in this case. `BufferedImage`’s constructor will throw a `RasterFormatException`
ception if the number and types of bands in the Raster do not match the number and types of bands required by the ColorModel.

Class Java2DExample (Fig. 4.13) provides a user interface for applying Java2D-ImageFilters to ImagePanels. Lines 23–26 declare the Java2DImageFilters. Lines 34–37 initialize the Java2DImageFilters. Lines 40–41 create imagePanel—the ImagePanel to be filtered. Lines 44–45 create filterMenu—the menu of Java2DImageFilters. Lines 52–54 create the first JMenuItem for filterMenu—originalMenuIItem. An ItemListener invokes ImagePanel’s display-Original method when originalMenuIItem is selected (lines 56–66). Lines 69–76 call method createMenuItem (lines 93–116) for each of the four Java2DImageFilters. This method creates a JMenuItem for the filter with the appropriate title and mnemonic. ImagePanel invokes method applyFilter when the JMenuItem is selected (line 108). Java2DExample contains method main (lines 119–125), for starting the application.

```java
// Java2DExample.java
// Java2DExample is an application that applies filters to an image using Java 2D.
package com.deitel.advjhtp1.java2d;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.lang.*;
import java.net.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

public class Java2DExample extends JFrame {

    private JMenuItem filterMenu;
    private ImagePanel imagePanel;

    // image filters
    private Java2DImageFilter invertFilter;
    private Java2DImageFilter sharpenFilter;
    private Java2DImageFilter blurFilter;
    private Java2DImageFilter colorFilter;

    // initialize JMenuItem
    public Java2DExample()
    {
        super( "Java 2D Image Processing Demo" );

        // create Java2DImageFilters
        blurFilter = new BlurFilter();
    }

    // other methods...
}
```

Fig. 4.13 Java 2D image-processing application GUI (part 1 of 4).
sharpenFilter = new SharpenFilter();
invertFilter = new InvertFilter();
colorFilter = new ColorFilter();

// initialize ImagePanel
imagePanel = new ImagePanel(
    Java2DExample.class.getResource( "images/ajhtp.png" ) );

// create JMenuBar
JMenuBar menuBar = new JMenuBar();
setJMenuBar( menuBar );

// create JMenu
filterMenu = new JMenu( "Image Filters" );
filterMenu.setMnemonic( 'I' );

// create JMenuItem for displaying original Image
JMenuItem originalMenuItem = new JMenuItem( "Display Original" );
originalMenuItem.setMnemonic( 'O' );

originalMenuItem.addActionListener(
    new ActionListener() {
      public void actionPerformed( ActionEvent action ) {
        imagePanel.displayOriginalImage();
      }
    } // end anonymous inner class
);

// create JMenuItem for Java2DImageFilters
JMenuItem invertMenuItem = createMenuItem( "Invert", 'I', invertFilter );
JMenuItem sharpenMenuItem = createMenuItem( "Sharpen", 'S', sharpenFilter );
JMenuItem blurMenuItem = createMenuItem( "Blur", 'B', blurFilter );
JMenuItem changeColorsMenuItem = createMenuItem( "Change Colors", 'C', colorFilter );

// add JMenuItem to JMenu
filterMenu.add( originalMenuItem );
filterMenu.add( invertMenuItem );
filterMenu.add( sharpenMenuItem );
filterMenu.add( blurMenuItem );
filterMenu.add( changeColorsMenuItem );

// add JMenu to JMenuBar
menuBar.add( filterMenu );

Fig. 4.13 Java 2D image-processing application GUI (part 2 of 4).
getContentPane().add( imagePanel, BorderLayout.CENTER );

// end Java2DExample constructor

// create JMenuItem and ActionListener for given filter
public JMenuItem createMenuItem( String menuItemName, char mnemonic, final Java2DImageFilter filter )
{
    // create JMenuItem
    JMenuItem menuItem = new JMenuItem( menuItemName );

    // set Mnemonic
    menuItem.setMnemonic( mnemonic );

    menuItem.addActionListener( new ActionListener()
    {
        // apply Java2DImageFilter when MenuItem accessed
        public void actionPerformed( ActionEvent action )
        {
            imagePanel.applyFilter( filter );
        }
    } // end anonymous inner class
    );

    return menuItem;
}

// start program
public static void main( String args[] )
{
    Java2DExample application = new Java2DExample();
    application.setDefaultCloseOperation( EXIT_ON_CLOSE );
    application.pack();
    application.setVisible( true );
}

Fig. 4.13 Java 2D image-processing application GUI (part 3 of 4).
This concludes our discussion of the Java 2D API. This section has presented several of the features that make Java 2D a powerful 2D graphics API. We discussed geometrical shape-rendering processes, including how to create and fill shapes with different colors and patterns, how to draw a `GeneralPath` and how to apply transforms to Java 2D shapes. We also introduced and discussed Java 2D image processing, including how to create and apply filters to `BufferedImages`.

Fig. 4.13  Java 2D image-processing application GUI (part 4 of 4).
4.4 Java 3D API

We live in a 3D world. Our vision enables us to see in three dimensions—x, y, and z coordinates. Many of the surfaces onto which graphics are displayed—for example, monitors and printed pages—are flat. 3D-graphics programming enables us to render realistic models of our 3D world onto a 2D-viewing surface. 3D graphics have advanced to the point that nearly anything you can see around you can be modeled—represented numerically by shape and size—and rendered—drawn on your computer screen.

There now exists an increasing number of 3D-computer-graphics applications—from flight simulators and medical-imaging equipment to 3D games and screen savers. Rapid advances in computer hardware have resulted in tremendous growth in the 3D-graphics industry. Developments in high-performance hardware led to developments in high-performance 3D graphics APIs—beginning in the 1970s with Siggraph’s CORE API, continuing in the 1980s with SGI’s OpenGL and on through today with Microsoft’s Direct3D and Java 3D™.

Sophisticated 3D graphics require sophisticated graphics algorithms that often involve complex math. However, the Java 3D API provides robust and advanced 3D-graphics capabilities to Java developers while hiding the mathematics behind graphics algorithms. Java 3D is a high-level graphics-programming API. Java 3D handles all the necessary low-level graphics calls, so developers can create high-performance 3D-graphics scenes without having to understand any underlying hardware. Like Java, Java 3D is write once run anywhere™. Java 3D applications will run in the same way across different 3D graphics platforms.

Sun Microsystems designed the Java 3D API with four major goals in mind—application portability, hardware independence, performance scalability and the ability to produce 3D graphics over a network. Simplifying of complex graphics operations played a key role in developing the Java 3D API. Some of the markets and applications for the Java 3D API include:

- 3D-data visualization
- collaborative applications
- gaming (especially network-based multiplayer systems)
- business graphics
- interactive educational systems
- molecular modeling and viewing (MCAD)
- 3D-Web development
- 3D-GUI development

Java 3D offers several features that these markets use to develop their 3D-applications:

- **Behavior**—Java 3D supports multiple types of behavior including animation and motion, *collision detection* (detecting when two objects collide) and *morphing* (transforming an image into another image).
- **Fog**—Java 3D supports fog content that restricts viewers' ability to see certain objects in the scene. For example, fog helps to create a realistic model of a rainstorm in a 3D game.
- **Geometry**—Java 3D has built-in 3D-geometric primitives for creating geometric shapes. Java 3D can render scenes generated by existing 3D authoring tools, such as 3DStudioMax, VRML and Lightwave3D.
- **Light**—Lights allow you to illuminate objects in a 3D scene. Java 3D supports different forms of light and control over color, direction and intensity.
- **Sound**—A unique feature of Java 3D is support for 3D sound.
- **Texture**—Java 3D supports *texture mapping* for attaching images over 3D-geometric models.

Next, we present an overview of the Java 3D API—we examine the structure of a Java 3D scene by presenting an application that incorporates 3D geometry, lights and interactive animation. In the next section, we explain how to obtain and install the Java 3D API so you can run the examples in this chapter and create your own 3D content.

### 4.4.1 Obtaining and Installing the Java 3D API

The Java 3D API requires that you have the *Java 2 Platform, Standard Edition* and either OpenGL or Direct3D installed on your computer—Java 3D uses OpenGL or Direct3D graphics libraries to render 3D scenes. You can obtain OpenGL from [www.opengl.org](http://www.opengl.org). You can obtain Direct3D—part of Microsoft’s *DirectX API*—from [www.microsoft.com/directx/](http://www.microsoft.com/directx/).

The Java 3D API is not integrated in the core Java 2 Platform. To use the Java 3D API, you must install the appropriate Java extension and utility packages. The Java 3D API packages differ slightly depending on which low-level graphics libraries are installed on your computer. The version of Java 3D used in this chapter requires the OpenGL graphics library and *Windows 2000* Operating System. The version of Java 3D packages you install depends on your operating system and graphics API. You can obtain the Java 3D packages and installation instructions from [java.sun.com/products/java-media/3D/download.html](http://java.sun.com/products/java-media/3D/download.html).

### 4.4.2 Java 3D Scenes

Pictures rendered with Java3D are called *scenes*. A scene—also called a *virtual universe*—is 3D space that contains a set of shapes. The root of the Java 3D scene is a *VirtualUniverse* object. The *VirtualUniverse* has a *coordinate system* for describing the location of *scene graphs* it contains. Each Java 3D scene is described by a number of scene graphs—hierarchical structures that specify attributes of a 3D environment. Each scene
graph attaches to the `VirtualUniverse` at a specified point in the `VirtualUniverse`'s coordinate system. A scene graph is composed of an internal coordinate system and a number of `branch graphs`. Each scene graph has an internal coordinate system, so developers can attach scene graphs with different coordinate systems in the same `VirtualUniverse`. Class `Locale` is the root node in a scene graph, which contains the attachment coordinate for the `VirtualUniverse` and a number of branch graphs. There are two types of branch graphs in Java 3D—`content-branch graphs` and `view-branch graphs`. Content-branch graphs specify content in 3D scenes, such as geometry, lighting, textures, fog and behaviors. View-branch graphs contain `viewing platforms`—collections of objects that specify the perspective, position, orientation and scale in 3D scenes. The viewing platform is also called the `viewpoint`.

The Java 3D class `SceneGraphObject` is the base class for all objects in a branch graph. A `SceneGraphObject` may contain a `Group`, which represents a node that contains multiple children. The children of a `Group` may be other `Groups`, `Leafs` or `NodeComponents`. `Leafs` specify geometry, lights and sound in content-branch graphs and the viewing-platform components in the view-branch graph. `NodeComponent` objects specify the various `components` of `Groups` and `Leafs` such as texture and coloring attributes. Figure 4.14 lists some Java 3D `Group`, `Leaf` and `NodeComponent` subclasses.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partial list of Java3D Group classes</strong></td>
<td></td>
</tr>
<tr>
<td>BranchGroup</td>
<td>A scene-graph’s root Node that attaches to a Locale.</td>
</tr>
<tr>
<td>Switch</td>
<td>Can render either a single child or a mask of children.</td>
</tr>
<tr>
<td>TransformGroup</td>
<td>Contains a single transformation (e.g., translation, rotation or scaling).</td>
</tr>
<tr>
<td><strong>Partial list of Java3D Leaf classes</strong></td>
<td></td>
</tr>
<tr>
<td>Behavior</td>
<td>Contains methods for gathering user input (e.g., key presses and mouse clicks) and describing objects’ behavior upon certain events (e.g., collisions).</td>
</tr>
<tr>
<td>Light</td>
<td>Describes a set of parameters for Java 3D light sources.</td>
</tr>
<tr>
<td>Shape3D</td>
<td>Describes 3D-geometric objects.</td>
</tr>
<tr>
<td>ViewPlatform</td>
<td>Controls the viewpoint for a 3D scene.</td>
</tr>
<tr>
<td><strong>Partial list of Java3D NodeComponent classes</strong></td>
<td></td>
</tr>
<tr>
<td>Appearance</td>
<td>Specifies Shape3D attributes, such as coloring and texture.</td>
</tr>
<tr>
<td>Material</td>
<td>Describes an illuminated object’s properties (e.g., reflective color and shini-ness).</td>
</tr>
<tr>
<td>Texture</td>
<td>Specifies properties for texture mapping—a technique for drawing 2D images over 3D geometric models.</td>
</tr>
</tbody>
</table>

*Fig. 4.14  Java 3D Group, Leaf and NodeComponent subclasses.*
4.4.3 A Java 3D Example

This section creates an interactive Java 3D scene. The application demonstrates how to create and use Java 3D Geometry and Lights. A Java Swing GUI enables the user to change the properties of the shapes and lights in the 3D scene. The application demonstrates mouse behaviors—i.e., using the mouse to rotate, scale and translate the 3D-shapes. The application consists of three classes—Java3DWorld (Fig. 4.15), ControlPanel (Fig. 4.21) and Java3DExample (Fig. 4.22). Figure 4.16–Fig. 4.20 show sample screen captures demonstrating the features of this application.

Class Java3DWorld (Fig. 4.15) creates the Java 3D environment using geometry, transforms and lighting. Lines 19–22 import the Java 3D utility packages which simplify the scene-content creation. Class Java3DWorld extends class Canvas3D (line 24), a java.awt.Canvas subclass for 3D rendering. We use a Canvas3D as the drawing surface for our 3D graphics application. Lines 26–38 declare the Java 3D objects we use in the application. We discuss each object’s function momentarily.

```java
// Java3DWorld.java
// Java3DWorld is a Java 3D Graphics display environment
// that creates a SimpleUniverse and provides capabilities for
// allowing a user to control lighting, motion, and texture
// of the 3D scene.
package com.deitel.advjhtp1.java3d;

// Java core packages
import java.awt.event.*;
import java.awt.*;
import java.net.*;

// Java extension packages
import javax.swing.event.*;
import javax.media.j3d.*;
import javax.vecmath.*;

// Java 3D utility packages
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.image.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.behaviors.mouse.*;

public class Java3DWorld extends Canvas3D {
    private Appearance appearance; // 3D object’s appearance
    private Box shape; // 3D object to manipulate
    private Color3f lightColor; // Light color
    private Light ambientLight; // ambient scene lighting
    private Light directionalLight; // directional light
    private Material material; // 3D objects color object
    private SimpleUniverse simpleUniverse; // 3D scene environment
    private TextureLoader textureLoader; // 3D object’s texture
```
private TransformGroup transformGroup;

private String imageName; // texture image file name

// Java3DWorld constructor
public Java3DWorld( String imageFileName )
{
    super( SimpleUniverse.getPreferredConfiguration() );

    imageName = imageFileName;

    // create SimpleUniverse (3D Graphics environment)
    simpleUniverse = new SimpleUniverse( this );

    // set default view point and direction
    ViewingPlatform viewPlatform =
        simpleUniverse.getViewingPlatform();

    viewPlatform.setNominalViewingTransform();

    // create 3D scene
    BranchGroup branchGroup = createScene();

    // attach BranchGroup to SimpleUniverse
    simpleUniverse.addBranchGraph( branchGroup );
}

// create 3D scene
public BranchGroup createScene()
{
    BranchGroup scene = new BranchGroup();

    // initialize TransformGroup
    transformGroup = new TransformGroup();

    // set TransformGroup's READ and WRITE permission
    transformGroup.setCapability( TransformGroup.ALLOW_TRANSFORM_READ );

    transformGroup.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE );

    // add TransformGroup to BranchGroup
    scene.addChild( transformGroup );

    // create BoundingSphere
    BoundingSphere bounds = new BoundingSphere(
        new Point3d( 0.0f, 0.0f, 0.0f ), 100.0 );

    appearance = new Appearance(); // create object appearance
    material = new Material();  // create texture material

Fig. 4.15 Creating a Java 3D SimpleUniverse with content (part 2 of 5).
appearance.setMaterial( material );

String rgb = new String( "RGB" );

// load texture for scene object
textureLoader = new TextureLoader(
    Java3DWorld.class.getResource( imageName ), rgb, this );

// set capability bits for enabling texture
textureLoader.getTexture().setCapability(
    Texture.ALLOW_ENABLE_WRITE );

// initial texture will not show
textureLoader.getTexture().setEnable( false );

// set object's texture
appearance.setTexture( textureLoader.getTexture() );

// create object geometry
Box shape = new Box( 0.3f, 0.3f, 0.3f,
    Box.GENERATE_NORMALS | Box.GENERATE_TEXTURE_COORDS,
    appearance );

// add geometry to TransformGroup
transformGroup.addChild( shape );

// initialize Ambient lighting
ambientLight = new AmbientLight();
ambientLight.setInfluencingBounds( bounds );

// initialize directionalLight
directionalLight = new DirectionalLight();

lightColor = new Color3f(); // initialize light color

// set initial DirectionalLight color
directionalLight.setColor( lightColor );

// set capability bits to allow DirectionalLight's
// Color and Direction to be changed
directionalLight.setCapability(
    DirectionalLight.ALLOW_DIRECTION_WRITE );

directionalLight.setCapability(
    DirectionalLight.ALLOW_DIRECTION_READ );

directionalLight.setCapability(
    DirectionalLight.ALLOW_COLOR_WRITE );

directionalLight.setCapability(
    DirectionalLight.ALLOW_COLOR_READ );

directionalLight.setInfluencingBounds( bounds );

Fig. 4.15 Creating a Java 3D SimpleUniverse with content (part 3 of 5).
```java
// add light nodes to BranchGroup
scene.addChild( ambientLight );
scene.addChild( directionalLight );

// initialize rotation behavior
MouseRotate rotateBehavior = new MouseRotate();
rotateBehavior.setTransformGroup( transformGroup );
rotateBehavior.setSchedulingBounds( bounds );

// initialize translation behavior
MouseTranslate translateBehavior = new MouseTranslate();
translateBehavior.setTransformGroup( transformGroup );
translateBehavior.setSchedulingBounds( new BoundingBox( new Point3d( -1.0f, -1.0f, -1.0f ),
    new Point3d( 1.0f, 1.0f, 1.0f ) ) );

// initialize scaling behavior
MouseZoom scaleBehavior = new MouseZoom();
scaleBehavior.setTransformGroup( transformGroup );
scaleBehavior.setSchedulingBounds( bounds );

// add behaviors to BranchGroup
scene.addChild( scaleBehavior );
scene.addChild( rotateBehavior );
scene.addChild( translateBehavior );

scene.compile();
return scene;
```

```java
// change DirectionLight color
public void changeColor( Color color )
{
    lightColor.set( color );
directionalLight.setColor( lightColor );
}
```

```java
// change geometry surface to textured image or material color
public void updateTexture( boolean textureValue )
{
textureLoader.getTexture().setEnable( textureValue );
}
```

```java
// change image used for texture
public void setImageName( String imageFileName )
{
    imageName = imageFileName;
}
```
The `Java3DWorld` constructor (lines 41–62) accepts as a `String` argument the image file for texture mapping. Class `SimpleUniverse`, which creates a Java 3D scene, encapsulates all the objects in the virtual universe and viewing platform. By using a `SimpleUniverse`, developers create and attach content-branch graphs—the `SimpleUniverse` uses this information to construct the 3D scene.

The first step in creating a Java 3D scene is to initialize the `Canvas3D` (line 43). The `Canvas3D` constructor takes as an argument a `java.awt.GraphicsConfiguration` (line 43). Method `getPreferredConfiguration` of class `SimpleUniverse` returns the system’s `java.awt.GraphicsConfiguration`, which specifies a graphics device, such as a computer monitor. Line 48 invokes the `SimpleUniverse` constructor, passing the `Canvas3D` as an argument. This constructor creates a Java 3D `SimpleUniverse` with the `Canvas3D` as the drawing surface. Class `SimpleUniverse` creates and configures the objects in the view branch graph. Lines 51–54 configure the viewing distance—the length between the viewer and the canvas—for our 3D scene. All objects in the view branch graph are members of class `ViewingPlatform`. Method `getViewingPlatform` of class `SimpleUniverse` returns a reference to the `ViewingPlatform` created inside the `SimpleUniverse` (lines 51–52). Method `setNominalViewingTransform` of class `ViewPlatform` sets the viewing distance for our 3D scene to the nominal (i.e., default) distance of \( \pi/4.0 \). We now create content for our Java 3D scene.

In this application, we add one content branch-graph to the `SimpleUniverse`. Line 57 calls method `createScene` (lines 65–172), which returns a content `BranchGroup`. Class `BranchGroup` is the root node of a scene graph in a Java 3D scene. The `BranchGroup` contains the children `Groups`, `Leaf`s and `NodeComponents` that describe the Java 3D scene. Line 60 attaches the content `BranchGroup` to the `SimpleUniverse` using method `addBranchGraph` of class `SimpleUniverse`.

Method `createScene` creates, constructs and compiles the `BranchGroup` content. Line 67 creates an instance of class `BranchGroup`. Line 70 creates a `TransformGroup`.

---

```java
193  // get image file name
194  public String getImageName()
195  {
196      return imageName;
197  }
198
199  // return preferred dimensions of Container
200  public Dimension getPreferredSize()
201  {
202      return new Dimension( 500, 500 );
203  }
204
205  // return minimum size of Container
206  public Dimension getMinimumSize()
207  {
208      return getPreferredSize();
209  }
210 }
```

Fig. 4.15 Creating a Java 3D `SimpleUniverse` with content (part 5 of 5).
Class **TransformGroup**—a subclass of **Group**—specifies transformational behavior such as rotation, scaling and translation. Lines 73–77 set the **READ** and **WRITE** capability bits for the **TransformGroup** using method **setCapabilityBits** of the **TransformGroup**. Capability bits are integer flags that specify whether a given object should allow its properties to be read or written during execution. Line 80 calls method **addChild** of class **BranchGroup**, which adds the **TransformGroup** to the **BranchGroup**.

**Performance Tip 4.1**

By default, Java 3D sets an object’s properties so they cannot be changed during run-time. Java 3D does this to increase run-time performance.

Lines 83–84 create a **BoundingSphere**. Class **BoundingSphere** creates a spherical bounding volume, which specifies the volume of space in which **Lights** and **Behaviors** affect geometry in the scene. Outside the bounding volume, the **Lights** and **Behaviors** have no impact on the scene’s geometry. Lines 83–84 create a **BoundingSphere** that is centered at the origin and has a 100 meter radius.

Line 86 creates the **Appearance** that describes the visual attributes of shapes. Lines 87 creates a default **Material** object. Class **Material** specifies the properties of an illuminated object—any object defined within the bounds of a **Light**. The default **Material** constructor specifies that objects in ambient white light will appear grey. The default **Material** constructor also enables any objects with the associated **Material** to be illuminated in the 3D scene. Line 88 calls method **setMaterial** of class **Appearance** to set the **Material** to the default material, although we could have created a **Material** object that would make the shape’s surface reflect like a mirror or shine like metal.

Lines 93–104 create and load the image for texture mapping. Class **com.sun.j3d.utils.image.TextureLoader** loads an **Image** for texturing. The **TextureLoader** constructor takes as arguments the image file (**imageName**), the image format (**rgb**) and an **ImageObserver**. Lines 97–98 invoke method **setCapability** of class **TextureLoader** with argument **ALLOW_ENABLE_WRITE** so the user can apply textures to the **Texture** object during execution. Every **TextureLoader** has an associated **Texture** object that contains the texturing attributes. Line 101 disables texture mapping using method **setEnable** of class **Texture**, although the user can enable it in runtime. Method **setTexture** of class **Appearance** sets the **Texture** object in **Appearance** to our **Texture** (line 104).

Lines 107–109 create a 3D **Box**—the shape that appears in our scene. The **Box** constructor takes as arguments the length, width and height, a set of integer flags that indicate the position information to generate and an **Appearance** object. Position information is generated when geometry is created—by default only spatial coordinates are generated. To ensure proper lighting and texture mapping for geometry, line 108 instructs the compiler to generate additional position information. Line 112 uses method **addChild** of class **TransformGroup** to add the **Box** to the **TransformGroup** so the user can perform transformations on the **Box**.

Line 115 creates an **AmbientLight** for the scene. Class **AmbientLight** is a uniform light source that illuminates all objects within its boundary. **AmbientLight** will not illuminate those objects outside its boundary. Line 116 calls method **setInfluencingBounds** to set the **AmbientLight** boundary using the **BoundingSphere** we created in line 86. Lines 119–140 create a **DirectionalLight** for the scene. Class **Direc-
tionalLight describes a light source that travels between two points—the source and destination. Line 119 creates a DirectionalLight using the default constructor. Line 121 creates a Color3f object—a color defined by three floats that represent the RGB color components. Line 124 calls method setColor of DirectionalLight to set the light source color. Lines 128–138 set the capability bits to allow the user to alter the color and direction of the DirectionalLight. Lines 143–144 add the two light sources to the BranchGroup. All objects in the BranchGroup will be illuminated—as long as these objects are enabled for illumination.

Lines 147–161 create different behaviors for the Box. We use MouseBehavior class in utility package com.sun.j3d.utils.behavior.mouse. Lines 147–149 create an instance of class MouseRotate, which stores a rotational transformation for an object controlled with the left mouse button. By moving the mouse while pressing the left mouse button, the user controls the rotation of the Box. Line 148 calls method setTransformGroup of class MouseRotate to gather the rotation information from the TransformGroup. Line 149 calls method setSchedulingBounds of MouseRotate to set MouseRotate’s bounding volume. Figure 4.16 shows the output when the user rotates the Box.

Class MouseTranslate—another subclass of MouseBehavior creates a behavior that controls the translation (i.e., the displacement) of shapes when the user presses the right mouse button, then drags the mouse. Line 152 creates an instance of MouseTranslate. Line 153 calls method setTransformGroup of class MouseTranslate to gather the translational information from the TransformGroup. Lines 154–156 call method setSchedulingBounds, passing as an argument a BoundingBox. Class BoundingBox creates a cubic boundary. BoundingBox’s constructors takes as arguments two Point3d objects, which represent the upper-right and lower-left vertices of the cube. Outside this BoundingBox, the MouseTranslate behavior does not work. Figure 4.17 shows the output when the user translates the Box.

Class MouseZoom—another subclass of MouseBehavior—controls the shape’s size when the user presses either the middle mouse button (on a three-button mouse) or the Alt key and left button (on a two-button mouse), then drags the mouse. Line 159 creates an instance of class mouseZoom. Line 160 calls method setTransformGroup of class MouseZoom to gather the scaling information from the TransformGroup. Line 161 calls method setSchedulingBounds, passing the BoundingSphere we created earlier in method createScene. Figure 4.18 demonstrates the output when the user scales the Box.

Lines 164–166 add the three MouseBehaviors to the BranchGroup. Line 168 calls method compile of class BranchGroup. Compiling a BranchGroup informs the Java 3D engine to optimize rendering the scene using the capability bits set by the developer.

To toggle texture mapping and lighting during execution, we implement methods that update the Appearance and DirectionalLight. Method changeColor (lines 175–179) uses a Color object to set the DirectionalLight color. Line 177 creates a Color3D object from the Color argument and passes it to method setColor of the DirectionalLight. Figure 4.19 shows the output as the user alters the color for DirectionalLight.

Method updateTexture (lines 182–185) toggles texture mapping of the shapes in the scene. This method takes a boolean argument that specifies whether to enable texture mapping for the 3D shape. Figure 4.20 shows the output when the user enables texture mapping.
Fig. 4.16  Demonstrating MouseRotate behavior.
Fig. 4.17 Demonstrating `MouseTranslate` behavior.
Fig. 4.18 Demonstrating MouseZoom behavior.
Fig. 4.19 Demonstrating changing color in Java 3D.
Fig. 4.20 Demonstrating texture mapping in Java 3D.
The user controls the \texttt{DirectionalLight} properties and texture mapping in the \texttt{Java3DWorld} using class \texttt{ControlPanel} (Fig. 4.21). Lines 18–21 declare three \texttt{JSliders} and one \texttt{JCheckbox} for the user to interact with the 3D application. Line 24 declares a reference to a \texttt{Java3DWorld} object to access its \texttt{updateTexture} and \texttt{changeColor} methods.

```
// ControlPanel.java
// ControlPanel is a JPanel that contains Swing controls
// for manipulating a Java3DWorld.
package com.deitel.advjhtp1.java3d;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class ControlPanel extends JPanel {
    // JSliders control lighting color
    private JSlider redSlider, greenSlider, blueSlider;
    // JCheckbox turns on texture mapping
    private JCheckBox textureCheckBox;
    // graphics display environment
    private Java3DWorld java3DWorld;

    // ControlPanel constructor
    public ControlPanel( Java3DWorld tempJ3DWorld )
    {
        java3DWorld = tempJ3DWorld;

        // assemble instruction panel
        JPanel instructionPanel = new JPanel();

        TitledBorder titledBorder =
            new TitledBorder( "Transformation Instructions" );

        titledBorder.setTitleJustification( TitledBorder.CENTER );
        instructionPanel.setBorder( titledBorder );

        JLabel rotationInstructions =
            new JLabel( "Rotation - Left Mouse Button",
                        SwingConstants.CENTER );

        JLabel translationInstructions =
            new JLabel( "Translation - Right Mouse Button",
                        SwingConstants.CENTER );

        ...
JLabel scalingInstructions =
    new JLabel( "Scale - Alt + Left Mouse Button",
        SwingConstants.CENTER );

    // add instruction JLabels to JPanel
    instructionPanel.add( rotationInstructions );
    instructionPanel.add( translationInstructions );
    instructionPanel.add( scalingInstructions );

    // assemble texture mapping control panel
    JPanel texturePanel = new JPanel();

    TitledBorder textureBorder =
        new TitledBorder( "Texture Controls" );

    textureBorder.setTitleJustification( TitledBorder.CENTER );
    texturePanel.setBorder( textureBorder );

    textureCheckBox = new JCheckBox( "Apply Texture Map to Image" );

    texturePanel.add( textureCheckBox );

    // create ItemListener for JCheckBox
    textureCheckBox.addItemListener(
        new ItemListener() {

            // invoked when checkbox selected/deselected
            public void itemStateChanged( ItemEvent event ) {
                if( event.getStateChange() == ItemEvent.SELECTED )
                    Java3DWorld.updateTexture( true );
                else
                    Java3DWorld.updateTexture( false );
            }
        } // end anonymous inner class
    );

    // create JPanel with instructionPanel and texturePanel
    JPanel topPanel = new JPanel( new GridLayout( 2, 1, 0, 20 ) );

    topPanel.add( instructionPanel );
    topPanel.add( texturePanel );

    // assemble lighting color control panel
    JPanel colorPanel = new JPanel( new FlowLayout( FlowLayout.LEFT, 15, 15 ) );

    TitledBorder colorBorder =
        new TitledBorder( "Direct Lighting Color Controls" );

Fig. 4.21  ControlPanel provides Swing controls for Java3DWorld (part 2 of 4).
colorBorder.setTitleJustification( TitledBorder.CENTER );
colorPanel.setBorder( colorBorder );

JLabel redLabel = new JLabel( "R" );
JLabel greenLabel = new JLabel( "G" );
JLabel blueLabel = new JLabel( "B" );

// create JSlider for adjusting red light component
redSlider = new JSlider(  
SwingConstants.HORIZONTAL, 0, 255, 25 );
redSlider.setMajorTickSpacing( 25 );
redSlider.setPaintTicks( true );

// create JSlider for adjusting green light component
greenSlider = new JSlider(  
SwingConstants.HORIZONTAL, 0, 255, 25 );
greenSlider.setMajorTickSpacing( 25 );
greenSlider.setPaintTicks( true );

// create JSlider for adjusting blue light component
blueSlider = new JSlider(  
SwingConstants.HORIZONTAL, 0, 255, 25 );
blueSlider.setMajorTickSpacing( 25 );
blueSlider.setPaintTicks( true );

// create ChangeListener for JSliders
ChangeListener slideListener = new ChangeListener() {  
  // invoked when slider has been accessed
  public void stateChanged(ChangeEvent event)  
  {  
    Color color = new Color(  
      redSlider.getValue(), greenSlider.getValue(),  
      blueSlider.getValue() );  
    
    Java3DWorld.changeColor( color );  
  }  
}; // end anonymous inner class

// add listener to sliders
redSlider.addChangeListener( slideListener );
greenSlider.addChangeListener( slideListener );
blueSlider.addChangeListener( slideListener );

// add lighting color control components to colorPanel
colorPanel.add( redLabel );
colorPanel.add( redSlider );
colorPanel.add( greenLabel );

Fig. 4.21 **ControlPanel** provides Swing controls for **Java3DWorld** (part 3 of 4).
There are three sets of controls for the Java3DWorld—transformation, texture mapping and lighting controls. The translations are controlled using MouseTranslate, MouseRotate and MouseZoom—no Swing components are needed to control the Java3DWorld transforms. Lines 32–55 create a JPanel that contains JLabels with instructions for applying transforms to the scene using the mouse.

Lines 58–69 create a JPanel with texture-mapping controls. The JCheckBox regulates the texture mapping in the application. Lines 72–85 attach an ItemListener to textureCheckBox. When the user selects this JCheckBox, line 79 calls method updateTexture of Java3DWorld to enable texture mapping. If the user deselects the JCheckBox, line 81 disables texture mapping.

Lines 104–127 create three JSliders that can assume an integer 0–255, inclusive. Lines 130–142 create aChangeListener for the JSliders. When the user accesses a JSlider, line 139 calls method changeColor of Java3DWorld to change the shape’s color in the 3D scene.

Class Java3DExample (Fig. 4.22) contains the Java3DWorld and ControlPanel. The Java3DExample constructor (lines 21–32) creates the Java3DWorld object by passing a String argument that specifies the image used for texture mapping (line 25). Line 26 creates controlPanel, passing the Java3DWorld as an argument. Method main (lines 35–41) executes the application.
4.5 A Java 3D Case Study: A 3D Game with Custom Behaviors

In Section 4.4.3, we demonstrated how the Java 3D MouseBehavior utility classes add interactive behavior to a 3D scene. The utility behavior classes provide a simple and convenient way to add interaction to our 3D applications. However, some applications—such as computer games—require custom behaviors (e.g., collision detection, navigation and position checking). In this section, we demonstrate how to implement custom behaviors using the javax.media.j3d.Behavior class. We demonstrate collision detection...
among scene obstacles, navigation through a 3D scene and position checking of a user-navigated shape to determine when it has reached its target. We also introduce how to animate shapes using **Interpolator**s. We examine how to use additional **Node** and **Group** subclasses, such as **Switches** and **Text3Ds**. The final product is a 3D game in which the user navigates a shape through a 3D scene full of “flying” obstacles. The goal of the game is to move this shape to a specific target point without having the shape collide with any of the moving obstacles.

Class **Java3DWorld1** (Fig. 4.23) creates the 3D game’s objects, behaviors and animation. Class **Java3DWorld1** extends class **Canvas3D** (line 22). Lines 25–54 declare constants for setting various parameters in our 3D scene. Line 56 declares the **Switch** that contains the flying shapes. Class **Switch** extends Java 3D class **Group**. A **Switch** object specifies which of its children to render. Line 57 declares the **BoundingSphere** for scheduling bounds for the scene graph. Line 59 declares the **SimpleUniverse** that contains the **Locale** and view branch graph for our application. Line 61 declares the **String** that describes the image file for texturing shapes.

The **Java3DWorld1** constructor (lines 64–85) accepts as an argument a **String** that represents the image file for texturing the target shape. Line 66 initializes the **Canvas3D** by invoking the superclass constructor with a **GraphicsConfiguration** argument. Method **getPreferredConfiguration** of class **SimpleUniverse** returns the system’s **java.awt.GraphicsConfiguration**, which specifies a graphics-output device. Line 71 creates a **SimpleUniverse** with the **Canvas3D** as the drawing surface. Class **SimpleUniverse** creates the Java 3D scene that encapsulates all the shapes in the virtual universe and viewing platform. Lines 74–77 configure the scene’s viewing distance to the default value (**PI/4.0**). Line 80 calls method **createScene** (lines 65–248) to create a **BranchGroup** that line 60 attaches to the **SimpleUniverse**.

Method **createScene** constructs the **BranchGroup** content. Line 93 creates the **Switch** object that contains the scenes in our 3D game by calling method **initializeSwitch** (lines 285–297). This method takes an **int** argument (**DEFAULT_SCENE**) that specifies the default scene to display upon creation. Lines 288–294 set the **Switch** group’s capability bits to allow the **Switch** (and its children) to be read and written at run time. In this application, we implement collision detection—lines 293–294 set the capability bit that allows collision reporting. Upon collision—i.e., when two shapes intersect—the Java 3D engine compiles the scene-graph path to the object that triggered the collision. When method **initializeSwitch** returns the **Switch** to method **createScene**, line 96 calls method **initializeSwitch** again to create a **Switch** that contains various shapes in the scene. Line 100 creates a **BranchGroup** that aggregates the shapes in **Switch** (line 103). Line 106 attaches the **BranchGroup** to the **Switch** of scenes. Line 109 attaches the **Switch** of scenes to the root **sceneBranchGraph**. Lines 112–113 create the **BoundingSphere** for setting the bounds for the **sceneBranchGraph**.

The 3D game features a scene with several 3D obstacles that “fly” across the screen. These obstacles rotate and translate at random. Rotation and translation are types of transformations. We discussed in the previous section that a **TransformGroup** holds the information about a spatial transformation and applies transformations to its children. The rotation and translation transformations animate the 3D shapes in our scene. Each shape needs two **TransformGroups**—one for rotation information and one for translation information. Lines 116–121 call method **createTransformGroupArray** (lines 294–
322) to create TransformGroup arrays spinTransform and pathTransform, which store the rotational and translation information, respectively. Method createTransformGroupArray takes an int argument (NUMBER_OF_SHAPES) that specifies the array size. Line 305 initializes each TransformGroup in the array, and lines 308–317 set the capability bits of each TransformGroup to enable collision reporting and allow reading and writing during run time.

Fig. 4.23 Class Java3DWorld1 creates the 3D-game environment (part 1 of 13).
public static final int MAX_PHASE_DELAY = 20000;

private static final float MAX_RADIUS = 0.15f;
private static final float MAX_LENGTH = 0.2f;
private static final float MAX_SHININESS = 128.0f;
private static final float SPHERE_RADIUS = 0.15f;
private static final float BOUNDING_RADIUS = 100.0f;

private Switch shapeSwitch; // contains flying shapes
private BoundingSphere bounds; // bounds for nodes and groups
private SimpleUniverse simpleUniverse; // 3D environment
private String imageName; // texture image file name

public Java3DWorld1( String imageFileName ) {
    super( SimpleUniverse.getPreferredConfiguration() );
    imageName = imageFileName;
    simpleUniverse = new SimpleUniverse( this );
    ViewingPlatform viewPlatform =
        simpleUniverse.getViewingPlatform();
    viewPlatform.setNominalViewingTransform();
    BranchGroup branchGroup = createScene();
    simpleUniverse.addBranchGraph( branchGroup );
}

public BranchGroup createScene() {
    BranchGroup sceneBranchGroup = new BranchGroup();
    Switch sceneSwitch = initializeSwitch( DEFAULT_SCENE );
    Switch shapeSwitch = initializeSwitch( DEFAULT_SCENE );
98 // initialize BranchGroup that contains only elements in game scene
99 BranchGroup gameBranchGroup = new BranchGroup();
100
101 // add shapeSwitch to gameBranchGroup
102 gameBranchGroup.addChild( shapeSwitch );
103
104 // add gameBranchGroup to sceneSwitch
105 sceneSwitch.addChild( gameBranchGroup );
106
107 // add sceneSwitch to sceneBranchGroup
108 sceneBranchGroup.addChild( sceneSwitch );
109
110 // create BoundingSphere for 3D objects and behaviors
111 bounds = new BoundingSphere(
112 new Point3d( 0.0f, 0.0f, 0.0f ), BOUNDING_RADIUS );
113
114 // create rotation TransformGroup array
115 TransformGroup[] spinTransform =
116 createTransformGroupArray( NUMBER_OF_SHAPES );
117
118 // create translation TransformGroup array
119 TransformGroup[] pathTransform =
120 createTransformGroupArray( NUMBER_OF_SHAPES );
121
122 // create RotationInterpolators
123 createRotationInterpolators( spinTransform,
124 NUMBER_OF_SHAPES );
125
126 // create PositionInterpolators
127 createPositionInterpolators( pathTransform,
128 NUMBER_OF_SHAPES );
129
130 // create Appearance objects for Primitives
131 Appearance[] shapeAppearance =
132 createAppearance( NUMBER_OF_SHAPES );
133
134 // create shapes
135 Primitive[] shapes =
136 createShapes( shapeAppearance, NUMBER_OF_SHAPES );
137
138 // add shapes to scene structure
139 for ( int x = 0; x < NUMBER_OF_SHAPES; x++ ) {
140
141 // add primitive to spinTransform group
142 spinTransform[ x ].addChild( shapes[ x ] );
143
144 // add spinTransform group to pathTransform group
145 pathTransform[ x ].addChild( spinTransform[ x ] );
146
147 // add pathTransform group to shapeSwitch group
148 shapeSwitch.addChild( pathTransform[ x ] );
149
150 }

Fig. 4.23 Class Java3DWorld1 creates the 3D-game environment (part 3 of 13).
// create and set scene lighting
setLighting( sceneBranchGroup, bounds );

// create scene to display if user loses
TransformGroup loserTransformGroup =
createEndScene( "You Lose!" );

// add loser scene to sceneSwitch
sceneSwitch.addChild( loserTransformGroup );

// create scene to display if user wins
TransformGroup winnerTransformGroup =
createEndScene( "You Win!" );

// add winner scene to sceneSwitch
sceneSwitch.addChild( winnerTransformGroup );

// create shiny red Appearance for navigating shape
Appearance flyingAppearance = createAppearance( new Color3f( 1.0f, 0.0f, 0.0f ) );

// initialize navigable sphere
Primitive flyingBall = new Sphere( 0.03f, Sphere.GENERATE_NORMALS, flyingAppearance );

// set capability bits to enable collision detection and
// allow for read/write of bounds
flyingBall.setCollidable( true );
flyingBall.setCapability( Node.ENABLE_COLLISION_REPORTING );
flyingBall.setCapability( Node.ALLOW_BOUNDS_READ );
flyingBall.setCapability( Node.ALLOW_BOUNDS_WRITE );

// create TransformGroup to translate shape position
TransformGroup startTransform = createTransform( new Vector3f( 0.9f, -0.9f, 0.0f ) );

startTransform.addChild( flyingBall );
gameBranchGroup.addChild( startTransform );

// create Material for Appearance for target sphere
Appearance targetAppearance = createAppearance( new Color3f( 0.0f, 1.0f, 0.0f ) );

// obtain textured image for target sphere
String rgb = new String( "RGB" );
TextureLoader textureLoader = new TextureLoader( Java3DWorld1.class.getResource( imageName ), rgb, this );
textureLoader.getTexture().setEnable( true );
targetAppearance.setTexture( textureLoader.getTexture() );
202 // initialize target sphere
203 Primitive targetSphere = new Sphere( SPHERE_RADIUS,
204     Sphere.GENERATE_TEXTURE_COORDS | Sphere.GENERATE_NORMALS,
205     targetAppearance );
206
207 // disable collision detection for sphere
208 targetSphere.setCollidable( false );
209
210 // create vector to target point
211 Vector3f target = new Vector3f( -1.0f, 1.0f, -1.0f );
212
213 // create TransformGroup that translates sphere position
214 TransformGroup targetTransform = createTransform( target );
215 targetTransform.addChild( targetSphere );
216 gameBranchGroup.addChild( targetTransform );
217
218 // create Navigator behavior
219 Navigator navigator = new Navigator( startTransform );
220 navigator.setSchedulingBounds( bounds );
221
222 // create Collide behavior
223 Collide collider = new Collide( simpleUniverse, flyingBall, sceneSwitch );
224 collider.setSchedulingBounds( bounds );
225
226 // create GoalDetector behavior
227 GoalDetector goalDetector = new GoalDetector( simpleUniverse, startTransform, sceneSwitch,
228     target, SPHERE_RADIUS );
229 goalDetector.setSchedulingBounds( bounds );
230
231 // add Behaviors to scene
232 sceneBranchGroup.addChild( goalDetector );
233 sceneBranchGroup.addChild( collider );
234 sceneBranchGroup.addChild( navigator );
235
236 // create Background for scene
237 Background background = new Background();
238 background.setColor( 0.4f, 0.4f, 1.0f );
239 background.setApplicationBounds( bounds );
240 sceneBranchGroup.addChild( background );
241
242 sceneBranchGroup.compile();
243
244 return sceneBranchGroup;
245 } // end method createScene
246
247 // create Appearance object for Primitive in scene
248 private Appearance createAppearance( Color3f diffuseColor )
249 {
250     Appearance appearance = new Appearance();
251     Material material = new Material();
252     appearance.setMaterial( material );
253     appearance.setDiffuseColor( diffuseColor );
254     return appearance;
255 } // end method createAppearance

Fig. 4.23 Class Java3DWorld1 creates the 3D-game environment (part 5 of 13).
material.setShininess( MAX_SHININESS );
material.setDiffuseColor( diffuseColor );
material.setAmbientColor( 0.0f, 0.0f, 0.0f );
appearance.setMaterial( material );
return appearance;

} // end method createAppearance

// create TransformGroup for placing an object in scene
private TransformGroup createTransform(
    Vector3f positionVector )
{
    // initialize a TransformGroup and set capability bits
    TransformGroup transformGroup = new TransformGroup();
    transformGroup.setCapability(
        TransformGroup.ALLOW_TRANSFORM_READ );
    transformGroup.setCapability(
        TransformGroup.ALLOW_TRANSFORM_WRITE );
    // translate starting position to bottom right of scene
    Transform3D location = new Transform3D();
    location.setTranslation( positionVector );
    transformGroup.setTransform( location );
    return transformGroup;
}

} // end method createTransform

// initialize Switch group and set capability bits
private Switch initializeSwitch( int sceneNumber )
{
    Switch switchGroup = new Switch( sceneNumber );
    switchGroup.setCollidable( true );
    switchGroup.setCapability( Switch.ALLOW_SWITCH_WRITE );
    switchGroup.setCapability( Switch.ALLOW_SWITCH_READ );
    switchGroup.setCapability( Group.ALLOW_CHILDREN_WRITE );
    switchGroup.setCapability( Group.ALLOW_CHILDREN_READ );
    switchGroup.setCapability( Group.ENABLE_COLLISION_REPORTING );
    return switchGroup;
}

} // end method initializeSwitch

private TransformGroup[] createTransformGroupArray(
    int size )
{
    TransformGroup[] transformGroup =
        new TransformGroup[ size ];
    // set TransformGroup's WRITE and READ permissions
    // and enable collision reporting

Fig. 4.23  Class Java3DWorld1 creates the 3D-game environment (part 6 of 13).
for ( int i = 0; i < size; i++ ) {
    // create TransformGroups
    transformGroup[ i ] = new TransformGroup();
    // enable collision reporting
    transformGroup[ i ].setCapability(
        Group.ENABLE_COLLISION_REPORTING );
    // enable WRITE permission
    transformGroup[ i ].setCapability(
        TransformGroup.ALLOW_TRANSFORM_WRITE );
    // enable READ permission
    transformGroup[ i ].setCapability(
        TransformGroup.ALLOW_TRANSFORM_READ );
}
return transformGroup;

// create RotationInterpolators for scene
private void createRotationInterpolators(
    TransformGroup[] transformGroup, int size )
{
    // declare structures for creating RotationInterpolators
    Alpha[] alphaSpin = new Alpha[ size ];
    Transform3D[] spinAxis = 
        new Transform3D[ size ];
    RotationInterpolator[] spinner =
        new RotationInterpolator[ size ];
    // create RotationInterpolator for each shape
    for ( int x = 0; x < size; x++ ) {
        // initialize Alpha
        alphaSpin[ x ] = new Alpha();
        // set increasing time for Alpha to random number
        double increasingAlphaDuration = MIN_ROTATION_SPEED + ((int)(Math.random() * (MAX_ROTATION_SPEED - MIN_ROTATION_SPEED) ));
        spinner[ x ] = new RotationInterpolator(
            increasingAlphaDuration, transformingGroup[ x ], transformGroup[ x ],
            spinAxis[ x ] = new Transform3D();

Fig. 4.23 Class Java3DWorld1 creates the 3D-game environment (part 7 of 13).
360  // set random X-axis rotation
361  spinAxis[ x ].rotX(
362      ( float ) ( Math.PI * ( Math.random() * 2 ) ) );
363  spinner[ x ].setAxisOfRotation( spinAxis[ x ] );
364
365  // set minimum and maximum rotation angles
366  spinner[ x ].setMinimumAngle( MIN_ROTATION_ANGLE );
367  spinner[ x ].setMaximumAngle( MAX_ROTATION_ANGLE );
368
369  spinner[ x ].setSchedulingBounds( bounds );
370
371  // add RotationInterpolator to appropriate TransformGroup
372  transformGroup[ x ].addChild( spinner[ x ] );
373  }
374
375  } // end method createRotationInterpolators
376
377  // create PositionInterpolators
378  private void createPositionInterpolators(
379      TransformGroup[] transformGroup, int size )
380  {
381      // create structures for PositionInterpolators
382      Alpha[] alphaPath = new Alpha[ size ];
383
384      PositionInterpolator[] mover =
385          new PositionInterpolator[ size ];
386
387      Transform3D[] pathAxis =
388          new Transform3D[ size ];
389
390      // create PositionInterpolator for each shape
391      for ( int x = 0; x < size; x++ ) {
392          // initialize Alpha
393          alphaPath[ x ] = new Alpha();
394
395          // set mode to increase and decrease interpolation
396          alphaPath[ x ].setMode(
397              Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE );
398
399          // set random phase delay
400          alphaPath[ x ].setPhaseDelayDuration(
401              ( ( int ) ( Math.random() * MAX_PHASE_DELAY ) ) );
402
403          // randomize translation speed
404          int speed = MIN_TRANSLATION_SPEED +
405              ( ( int ) ( Math.random() * MAX_TRANSLATION_SPEED ) );
406
407          // set increasing and decreasing durations
408          alphaPath[ x ].setIncreasingAlphaDuration( speed );
409          alphaPath[ x ].setDecreasingAlphaDuration( speed );
410
Fig. 4.23  Class Java3DWorld1 creates the 3D-game environment (part 8 of 13).
// randomize translation axis
pathAxis[ x ] = new Transform3D();
pathAxis[ x ].rotX(
    ( float ) ( Math.PI * ( Math.random() * 2 ) ) );
pathAxis[ x ].rotY(
    ( float ) ( Math.PI * ( Math.random() * 2 ) ) );
pathAxis[ x ].rotZ(
    ( float ) ( Math.PI * ( Math.random() * 2 ) ) );

// initialize PositionInterpolator
mover[ x ] = new PositionInterpolator( alphaPath[ x ],
    transformGroup[ x ], pathAxis[ x ], 1.0f, -1.0f );

mover[ x ].setSchedulingBounds( bounds );

// add PositionInterpolator to appropriate TransformGroup
transformGroup[ x ].addChild( mover[ x ] );
}
} // end method createPositionInterpolators

// create appearance and material arrays for Primitives
private Appearance[] createAppearance( int size )
{
    // create Appearance objects for each shape
    Appearance[] appearance =
        new Appearance[ size ];

    Material[] material = new Material[ size ];

    // set material and appearance properties for each shape
    for( int i = 0; i < size; i++ ) {
        appearance[ i ] = new Appearance();
        material[ i ] = new Material();

        // set material ambient color
        material[ i ].setAmbientColor( new Color3f( 0.0f, 0.0f, 0.0f ) );

        // set material Diffuse color
        material[ i ].setDiffuseColor( new Color3f( ( float ) Math.random(), ( float ) Math.random(),
            ( float ) Math.random() ) );

        // set Material for appropriate Appearance object
        appearance[ i ].setMaterial( material[ i ] );
    }

    return appearance;
}
} // end method createAppearance

Fig. 4.23 Class Java3DWorld1 creates the 3D-game environment (part 9 of 13).
private Primitive[] createShapes( Appearance[] appearance, int size ) {
    Primitive[] shapes = new Primitive[size];

    // random loop to get index
    for ( int x = 0; x < size; x++ ) {
        // generate random shape index
        int index = (int) (Math.random() * NUMBER_OF_PRIMITIVES);

        // create shape based on random index
        switch( index ) {
            case 0: // create Box
                shapes[ x ] = new Box(
                    ( float ) Math.random() * MAX_LENGTH,
                    ( float ) Math.random() * MAX_LENGTH,
                    ( float ) Math.random() * MAX_LENGTH,
                    Box.GENERATE_NORMALS, appearance[ x ]
                );
                break;

            case 1: // create Cone
                shapes[ x ] = new Cone(
                    ( float ) Math.random() * MAX_RADIUS,
                    ( float ) Math.random() * MAX_LENGTH,
                    Cone.GENERATE_NORMALS, appearance[ x ]
                );
                break;

            case 2: // create Cylinder
                shapes[ x ] = new Cylinder(
                    ( float ) Math.random() * MAX_RADIUS,
                    ( float ) Math.random() * MAX_LENGTH,
                    Cylinder.GENERATE_NORMALS, appearance[ x ]
                );
                break;

            case 3: // create Sphere
                shapes[ x ] = new Sphere(
                    ( float ) Math.random() * MAX_RADIUS,
                    Sphere.GENERATE_NORMALS, appearance[ x ]
                );
                break;

        } // end switch statement

        // set capability bits to enable collisions and to set
        // read/write permissions of bounds
        shapes[ x ].setCapability(Node.ENABLE_COLLISION_REPORTING);
        shapes[ x ].setCapability(Node.ALLOW_BOUNDS_READ);
        shapes[ x ].setCapability(Node.ALLOW_BOUNDS_WRITE);
    }

    return shapes;
}
shapes[ x ].setCollidable( true );
}

return shapes;

} // end method createShapes

// initialize ambient and directional lighting
private void setLighting( BranchGroup scene,
    BoundingSphere bounds )
{
    // initialize ambient lighting
    AmbientLight ambientLight = new AmbientLight();
    ambientLight.setInfluencingBounds( bounds );

    // initialize directional lighting
    DirectionalLight directionalLight = new DirectionalLight();
    directionalLight.setColor( new Color3f( 1.0f, 1.0f, 1.0f ) );
    directionalLight.setInfluencingBounds( bounds );

    // add lights to scene
    scene.addChild( ambientLight );
    scene.addChild( directionalLight );
}

} // end method setLighting

// update scene by rendering different shapes in shapeSwitch
public void switchScene( int numberChildren, int size )
{
    // create a new BitSet of size NUMBER_OF_SHAPES
    BitSet bitSet = new BitSet( size );

    // set BitSet values
    for ( int i = 0; i < numberChildren; i++ )
        bitSet.set( i );

    // instruct switchShape to render Mask of objects
    shapeSwitch.setWhichChild( Switch.CHILD_MASK );
    shapeSwitch.setChildMask( bitSet );

} // end method switchScene

// create end scene when user wins or loses
private TransformGroup createEndScene( String text )
{
    TransformGroup transformGroup = new TransformGroup();
    transformGroup.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE );

    // disable scene collision detection
    transformGroup.setCollidable( false );
// create Alpha object
Alpha alpha = new Alpha();
    alpha.setIncreasingAlphaDuration( MAX_ROTATION_SPEED );

// create RotationInterpolator for scene
RotationInterpolator rotation =
    new RotationInterpolator( alpha, transformGroup );

// set axis of rotation
Transform3D axis = new Transform3D();
    axis.rotY( ( float ) ( Math.PI / 2.0 ) );
    rotation.setAxisOfRotation( axis );

// set minimum and maximum rotation angles
rotation.setMinimumAngle( 0.0f );
rotation.setMaximumAngle( ( float ) ( Math.PI * 8.0 ) );

rotation.setSchedulingBounds( bounds );
transformGroup.addChild( rotation );

// create scene geometry
Appearance appearance = new Appearance();
Material material = new Material();
appearance.setMaterial( material );

// set diffuse color of material
material.setDiffuseColor( new Color3f( 0.0f, 0.8f, 1.0f ) );

// create Font3D object
Font3D font3d = new Font3D( new Font( "Helvetica", Font.ITALIC, 1 ),
    new FontExtrusion() );

// create Text3D object from Font3D object
Text3D text3d = new Text3D( font3d, text,
    new Point3f( -2.0f, 0.0f, 0.0f ) );

// create Shape3D object from Text3D object
Shape3D textShape = new Shape3D( text3d );
textShape.setAppearance( appearance );

// disable collision detection
textShape.setCollidable( false );
transformGroup.addChild( textShape );

    return transformGroup;

} // end method createEndScene

Fig. 4.23 Class Java3DWorld1 creates the 3D-game environment (part 12 of 13).
When method `createTransformGroupArray` returns the array, method `createScene` creates the `Interpolator`s, which help animate the transformations. Class `Interpolator` is a subclass of `Behavior`—later, we discuss how to implement custom behaviors. Now, we discuss how to use some well-known `Behavior` subclasses provided by Java 3D. To animate shapes smoothly in a scene, Java 3D provides `Interpolator`s and `Alpha`s. `Interpolator`s use `Alpha`s to specify certain characteristics of animation, such as the speed of transformations (e.g., rotation speed), or how fast a shape changes color (e.g., lighting effects). `Interpolator` objects convert time values to transformations of 3D shapes—`Alpha` objects generate these time values. For example, when a shape “flies” across the screen in one second, the `Interpolator` converts `Alpha`-generated time values (from 0 to 1) to translation operations that move the shape in the one-second period. An `Interpolator` operates in conjunction with a `Trans-
formGroup—each TransformGroup has an associated Interpolator. The Interpolator describes how to animate shapes in the TransformGroup. Java 3D provides several Interpolator subclasses. In this game, we use RotationInterpolator for rotating objects and PositionInterpolator for translating objects.

Lines 124–125 pass the spinTransform array of TransformGroups to method createRotationInterpolators (lines 330–375), which initializes the RotationInterpolator for spinTransform. An Alpha object and a Transform3D object compose a RotationInterpolator object. Alpha objects contain a series of phases that either increase or decrease. Increasing Alpha objects generate values in a sequence from 0 to 1, whereas decreasing Alpha objects generate values in a sequence from 1 to 0. An Alpha object’s default constructor sets that Alpha to generate increasing values, which result in a shape spinning in one specific direction (decreasing values enable the shape to spin in the opposite direction). For each Alpha object, lines 349–351 call method setIncreasingAlphaDuration, which specifies the time (in milliseconds) for that Alpha object to increase from 0 to 1. A random-number generation sets the time value between MIN_ROTATION_SPEED and MAX_ROTATION_SPEED. Lines 355–356 create the RotationInterpolator array that use the Alpha array and the spinTransform array. The Alpha object controls the Interpolator, which in turn transforms the 3D shapes in the TransformGroup. The RotationInterpolator constructor creates a default Transform3D for the rotation. Class Transform3D is a two-dimensional array that represents a general transform—in this case, a rotation. Each Transform3D has an associated integer type that determines the transformation to represent. Lines 358–369 create the Transform3D for the RotationInterpolator. Lines 361–363 assign the Transform3D a random axis of rotation. Lines 366–367 assign the minimum and maximum rotation angles (i.e., the starting and stopping angles for a complete rotational period). Line 372 adds the RotationInterpolators to the TransformGroup spinTransform.

When method createRotationInterpolators returns, lines 128–129 pass pathTransform as an argument to method createPositionInterpolators (lines 378–431). This method creates a set of PositionInterpolators that translate the shapes in the scene—specifically, the method creates PositionInterpolators for each TransformGroup in pathTransform. PositionInterpolators operate similarly to RotationInterpolators, except PositionInterpolators translate a 3D shape’s position on a given axis, whereas RotationInterpolators rotate a 3D shape on a given axis. Line 382 creates the Alpha objects associated with the PositionInterpolators—these values provide the time values that help to determine the shapes’ position. Lines 397–398 set the Alpha object as increasing and decreasing to ensure that the 3D shapes move back and forth across the screen. If the Alpha object was only decreasing or increasing, the shapes would move in only one direction. We chose to delay the initial movement of each shape to ensure that the 3D obstacle will not collide with the user-navigated shape immediately after the player starts the game. Lines 401–402 accomplish this by setting a randomized phase delay on each Alpha object. To make the game more interesting, we chose to set the increasing and decreasing durations to random speeds. Lines 413–419 assign random translation axes to the Transform3D objects that hold translation information to give obstacles different directions. Lines 422–423 pass five arguments to the PositionInterpolator constructor. The first three
arguments are the array of Alpha values, the array of TransformGroup values (path-Transform) and the array of Transform3D values. The last two arguments specify the starting and ending positions in the 3D scene for the PositionInterpolator translation. Line 428 adds each PositionInterpolator to each TransformGroup in pathTransform.

At this point, Java3DWorld1 has created the TransformGroups and Interpolators for each 3D shape. Now Java3DWorld1 must create the actual shapes. Lines 132–133 invoke method createAppearance (lines 434–461), which creates an array of randomly colored Appearance objects. Line 440 creates an array of Material objects, because every Appearance object has an associated Material object. Lines 448–449 sets each Material’s ambient color—the Material’s color when illuminated by reflected light. Lines 452–454 randomly set each Material’s diffuse color—the Material’s color when illuminated by some light source. Line 457 sets each Material in the Material array to an associated Appearance in the Appearance array. Line 459 then returns the Appearance array.

Lines 136–137 pass the Appearance array to method createShapes (lines 464–522) to create an array of Primitives that represent the shapes of the obstacles. The Java 3D com.sun.j3d.utils.geometry package provides four types of 3D-geometric Primitive types: Box, Cone, Cylinder and Sphere. Line 473 randomly generates a number between 0 and 3—each number is associated with one of these Primitive object. Lines 476–506 implement a switch statement—each case creates a unique type. The constructor of each Primitive subclass specifies that Primitive’s dimensions, lighting and appearance. Consider the Box constructor—lines 479–483 pass the Box’s length, width and height, the GENERATE_NORMALS constant (for the direction of the lighting), and an Appearance object. Lines 510–515 set the capability bits for each Primitive to enable collision reporting and read/write access during execution. Line 516 invokes method setCollidable of each Primitive’s Node superclass, so each Primitive can collide with other “collidable” Primitives. Line 520 returns the array of 3D shapes.

Lines 140–150 set up the sceneBranchGraph. Line 143 adds each 3D shape to each TransformGroup in spinTransform, line 146 adds each TransformGroup in spinTransform to each TransformGroup in pathTransform. Line 149 adds each TransformGroup in pathTransform to shapeSwitch. Line 153 calls method setLighting (lines 525–542) to create the AmbientLight and Directional-Light that illuminate the shapes in the scene. Lines 156–157 call method createEndScene (lines 561–620) to create a TransformGroup associated with the player losing the game. Method createEndScene uses its String argument to create a rotating object of class Text3D—a Geometry subclass for representing 3D text. Lines 563–568 create the TransformGroup to hold the Text3D. Lines 571–587 create the RotationInterpolator for rotating the Text3D. Lines 591–594 create an Appearance object for the Text3D. Lines 600–602 create a Font3D object, which uses both a java.awt.Font object and a Java 3D FontExtrusion object. A FontExtrusion describes the adding of a third dimension to the Font’s 2D text. Using the Font3D object, the String argument that holds the text and a Point3f object—x-y-z coordinates that specify a location in a SimpleUniverse, lines 605–606 create the Text3D object. Line 609 creates a Shape3D—a Node that describes a 3D shape—from the Text3D. Line 611
sets the `Shape3D`’s Appearance. Line 614 specifies that the `Shape3D` objects in this scene should not collide with other `Shape3D` objects. Line 616 adds each `Shape3D` to the `TransformGroup`, and line 619 returns the `TransformGroup`. Line 160 adds this `TransformGroup` (for the losing scene) to the `Switch` group. When the user-navigated shape collides with an obstacle, the application displays a scene with the rotating 3D text “You Lose.” Lines 163–164 call method `createEndScene` to create the scene that displays “You Win” when the player navigates the shape to the destination without collision. Line 167 adds this scene (`TransformGroup`) to the `Switch`.

The two missing pieces in our game are the navigable shape and the target (destination) shape. Lines 170–189 create a shiny red `Sphere` as the shape that the user navigates to the target shape. Lines 192–193 call method `createAppearance` (lines 251–261) to set this shape’s Appearance. This method takes as an argument a `Color3f` object and initializes an Appearance object based on a `Material` object that uses the `Color3f` object. Lines 174–175 instantiate the navigable shape as a `Sphere`. Lines 179–182 enable this `Sphere` to collide with the other `Primitives` in the scene. Lines 185–186 call method `createTransform` to create a `TransformGroup` to translate the `Sphere`’s starting position to the bottom-right corner of the scene. Lines 188–189 add the `Sphere` to this `TransformGroup`, then add the `TransformGroup` to the `gameBranchGroup`. Lines 192–216 create the game’s target shape: a `Sphere` that contains an image texture. Lines 196–200 load an image in a `Texture` object, then create an `Appearance` object with this `Texture` object. Lines 203–205 instantiate the target `Sphere`, and line 208 ensures that the `Sphere` cannot collide with the other `Primitives` in the game. We discuss later how the user-navigated shape interacts with the target `Sphere` (i.e., how the user-navigated shape determines that it has reached its goal). Line 214 calls method `createTransform` to create a `TransformGroup` that places the target `Sphere` in the upper-right corner of the scene. Line 215 adds the `Sphere` to this `TransformGroup`, and line 216 adds the `TransformGroup` to the `gameBranchGroup`.

We designed the game so the user can control the game difficulty. Using a `JSlider` in class `ControlPanel`, the user can specify the number of obstacles in the game. Method `switchScene` (lines 545–558) accepts an `int` argument that represents the number of shapes to display. Lines 548–552 create a `BitMask` from the `int` argument, then lines 555–556 renders each shape associated with the `BitMask`.

The last step in creating `Java3DWorld1` involves implementing a set of custom behaviors—that is, collision detection, navigation and goal detection. Lines 219–231 create these three behaviors. Class `Collide` enables shapes to detect collision, class `Navigator` enables the user (using the keyboard) to navigate the shiny red `Sphere` through the scene and class `GoalDetector` helps determine when this `Sphere` has reached the target `Sphere`. We discuss each class in detail momentarily. We add these behaviors to the `sceneBranchGraph`. Line 244 compiles `sceneBranchGraph` to create the displayable 3D scene.

### Custom Behaviors

The previous section demonstrated `Interpolators`—a set of `Behavior` subclasses that specify certain animation characteristics. Developers often need more specialized behaviors for 3D applications (e.g., collision detection, navigation and position checking). The Java 3D API provides the abstract `Behavior` class to create these custom behaviors. A `Behavior` object has an associated `behavior scheduler` responsible for registering
wake-up conditions—criteria that determines when the behavior scheduler should trigger a behavior. The behavior scheduler is a Java 3D subsystem that shields developers from implementation details. The behavior scheduler registers the wake-up conditions and handles the logic for when these conditions are satisfied. All classes that extend `Behavior` must implement methods `initialize` and `processStimulus`. Method `initialize` registers a set of wake-up conditions with the behavior scheduler. Method `processStimulus` handles the logic when the wake-up conditions are satisfied. The developer must implement method `processStimulus`, although typically, `processStimulus` determines the wake-up conditions that caused the event, handle the event (e.g., modify the scene-graph, etc.) and then reregister the wake-up conditions with the behavior scheduler.

The application in this section demonstrates three types of custom behavior: collision detection, navigation and position checking. We begin with the collision-detection behavior.

Collision detection determines when a shape’s bounding volume—the volume enclosing either a shape or the bounds of a shape—intersects another. Class `Collide` (Fig. 4.24), which extends superclass `Behavior`, implements collision-detection behavior for our Java 3D application. In Java 3D, shapes are either armed nodes or triggering nodes. A collision occurs when an armed node’s bounding volume intersects a triggering node’s bounding volume. Line 21 declares the armed node for collision detection. Line 24 declares the `WakeupCondition` object for our `Behavior` class. A Java 3D `Behavior` object passes the `WakeupCondition` to the behavior scheduler. When the `WakeupCondition` is satisfied (i.e., upon collision), the behavior scheduler returns an enumeration of the `WakeupCriterion` that triggered the behavior. Line 26 declares the `Switch` that contains scenes for the `SimpleUniverse` to display. Line 27 declares a reference to the Java 3D `SimpleUniverse` for displaying scenes in the `Switch`.

The `Collide` constructor (lines 33–52) takes a `SimpleUniverse`, a `Node` and a `Switch` as arguments. The reference to the `SimpleUniverse` adjusts the `ViewPlatform` when displaying different scenes in the `Switch`. The `Node` is the arming node for the collision-detection behavior. Lines 41–43 initialize `WakeupOnCollisionEntry`—the specific `WakeupCriterion` for our `Behavior` class. Class `WakeupOnCollisionEntry` takes as arguments an arming `Node` and integer `USE_GEOMETRY`, which specifies the `Node`’s geometric volume as the bounding surface for collision detection. Line 46 initializes the array of `WakeupCriterion` for our behavior class. This array contains only one element—`WakeupOnCollisionEntry`. Line 50 initializes the `WakeupCondition` as a `WakeupOr` that contains the `WakeupCriterion`. The objects in `WakeupOr` generate events when a `WakeupCriterion` is satisfied (when a collision occurs). Method `initialize` (lines 55–59) registers the `WakeupCondition` with the behavior scheduler by calling method `wakeupOn` of superclass `Behavior`. Method `wakeupOn` takes as an argument the `WakeupCondition` object, which registers with the behavior scheduler.

Upon collision, the behavior scheduler calls method `processStimulus` (lines 62–81), passing as an argument an `Enumeration` of the `WakeupCriteria` that triggered the event. Lines 65–79 handle each `WakeupCriterion` in the `Enumeration`. Line 72 handles only those `WakeupCriteria` that are `WakeupOnCollisionEntry` events. Line 73 invokes method `processCollision` (lines 85–106) for those `WakeupCriteria` that satisfy this condition. Line 77 reregisters the `WakeupCriterion` with the behavior scheduler.
Method `processCollision` handles the logic in response to the collision. In this application, a collision implies that the armed node (i.e., the user-navigated shape) has collided with an obstacle—the user then loses the game. Lines 87–100 set the translation component of the `ViewPlatform`'s `Transform3D`—the camera shifts back to expand the view. Line 104 switches to the scene associated with a collision—rotating 3D text that reads “You Lose.” Figure 4.24 demonstrates the 3D application display after the user-navigated shape collides with an obstacle.

```java
package com.deitel.advjhtp1.java3dgame;

import java.lang.*;
import java.util.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;

public class Collide extends Behavior {

    private Node armingNode;

    private WakeupCondition wakeupCondition;

    private Switch switchScene; // Switch group contains 3D scenes

    private SimpleUniverse simpleUniverse;

    private static final int LOSER_SCENE = 1;

    public Collide( SimpleUniverse universe, Node node, Switch tempSwitch )
    {
        armingNode = node;
        switchScene = tempSwitch;
        simpleUniverse = universe;

        WakeupOnCollisionEntry wakeupEvent =
                new WakeupOnCollisionEntry( armingNode, WakeupOnCollisionEntry.USE_GEOMETRY );
```
// set of WakeupEvents to which Behavior responds
WakeupCriterion[] wakeupCriteria = { wakeupEvent };

// Behavior responds when any WakeupEvent in
// WakeupCriterion occurs
wakeupCondition = new WakeupOr( wakeupCriteria );

} // end constructor

// initialize Behavior's wakeup conditions
public void initialize()
{
    // register WakeupCriterion to respond to collision events
    wakeupOn( wakeupCondition );
}

// handle WakeupEvents
public void processStimulus( Enumeration detected )
{
    // loop to handle events
    while( detected.hasMoreElements() ) {

        // get next sequential element
        WakeupCriterion criterion =
            ( WakeupCriterion ) detected.nextElement();

        // process event if WakeupOnCollisionEntry
        if ( criterion instanceof WakeupOnCollisionEntry ) {
            processCollision();

            // re-register WakeupCriterion to respond to new
            // WakeupOnCollisionEntry event
            wakeupOn( wakeupCondition );
        }
    }
} // end method processStimulus

// process collision by moving camera view back and
// switching scenes in Switch group
private void processCollision()
{
    Transform3D shiftViewBack = new Transform3D();

    // set Transform3D's Translation
    shiftViewBack.setTranslation(
        new Vector3f( 0.0f, 0.0f, 8.0f ) );

    // set Transform3D that determines View
    ViewingPlatform viewPlatform =
        simpleUniverse.get ViewingPlatform();
We provide class **Navigator** (Fig. 4.25) so the user can navigate the shape in our 3D scene. Class **Navigator** responds to certain key presses by translating a **Node** in a 3D scene. **Navigator** moves the **Node** by updating that **Node**'s **TransformGroup**. Line 22 declares the **TransformGroup**. Lines 25–30 declare **float** constants that represent the amount by which **Navigator** translates the shape upon each keypress. Line 33 declares the **WakeupCondition** for activating the navigational behavior.

```java
TransformGroup platformTransform =
    viewPlatform.getViewPlatformTransform();
platformTransform.setTransform( shiftViewBack );

// render scene in Switch group
switchScene.setWhichChild( LOSER_SCENE );
```

**Fig. 4.24** Implementing collision detection in a Java 3D application (part 3 of 3).

**Fig. 4.25** Behavior that enables the user to navigate a 3D shape (part 1 of 4).
public class Navigator extends Behavior {

private TransformGroup objectTransform;

// translation amounts
private static final float LEFT = -0.02f;
private static final float RIGHT = 0.02f;
private static final float UP = 0.02f;
private static final float DOWN = -0.02f;
private static final float FORWARD = 0.02f;
private static final float BACKWARD = -0.02f;

private WakeupCondition wakeupCondition;

public Navigator( TransformGroup transform )
{
    objectTransform = transform;

    // initialize WakeupOnAWTEvent to respond to
    // AWT KeyEvent.KEY_PRESSED events
    WakeupOnAWTEvent wakeupEvent =
        new WakeupOnAWTEvent( KeyEvent.KEY_PRESSED );

    // set of WakeupEvents to which Behavior responds
    WakeupCriterion[] wakeupCriteria = { wakeupEvent };
```java
// handle WakeupEvents
public void processStimulus( Enumeration detected )
{
    // loop to handle events
    while ( detected.hasMoreElements() ) {
        // get next WakeupCriterion
        WakeupCriterion wakeupCriterion =
            ( WakeupCriterion ) detected.nextElement();

        // handle WakeupCriterion if WakeupOnAWTEvent
        if ( wakeupCriterion instanceof WakeupOnAWTEvent ) {
            WakeupOnAWTEvent awtEvent =
                (WakeupOnAWTEvent) wakeupCriterion;
            AWTEvent[] events = awtEvent.getAWTEvent();

            // invoke method moveObject with AWTEvent
            moveShape( events );
        }
    }

    // re-register wakeupCondition to respond to next key press
    wakeupOn( wakeupCondition );
}

// handle AWT KeyEvents by translating an object in 3D scene
private void moveShape( AWTEvent[] awtEvents )
{
    // handle all events in AWTEvent array
    for ( int x = 0; x < awtEvents.length; x++)
    {
        // handle if AWTEvent is KeyEvent
        if ( awtEvents[ x ] instanceof KeyEvent ) {
            KeyEvent keyEvent = ( KeyEvent ) awtEvents[ x ];

            // respond only if KeyEvent is of type KEY_PRESSED
            if ( keyEvent.getID() == KeyEvent.KEY_PRESSED ) {
                // get KeyCode associated with KeyEvent
                int keyCode = keyEvent.getKeyCode();

                Transform3D transform3D = new Transform3D();

                // get Transform3D from TransformGroup of
                // navigable object
                objectTransform.getTransform( transform3D );

                Vector3f translateVector = new Vector3f();

    Fig. 4.25 Behavior that enables the user to navigate a 3D shape (part 3 of 4).
```
The `Navigator` constructor (lines 36–52) accepts as an argument a `TransformGroup` that contains the navigable 3D shape. Lines 42–43 initialize a `WakeupOnAWTEvent` that triggers a `Behavior` upon an `AWTEvent` (such as a keypress). The
The `WakeupOnAWTEvent` constructor takes as an argument the specific `AWTEvent` satisfies the wake-up conditions. In this case, `KeyEvent.KEY_PRESSED` events activate `Behavior`. Line 46 creates the `WakeupCriterion`s from the `WakeupOnAWTEvent`. Line 50 creates the `WakeupOr` that contains the `WakeupCriteria`.

Method `initialize` (lines 55–60) registers the `WakeupOr` with the behavior scheduler by passing the `WakeupOr` to method `wakeupOn` of superclass `Behavior`. Method `processStimulus` (lines 63–86) handles the logic for the triggered `Behavior` (i.e., a keypress). This method takes as an argument an `Enumeration` of the `WakeupCriterion` objects associated with the `Behavior`. Lines 69–70 retrieve each `WakeupCriterion` from the `Enumeration`. If the `WakeupCriterion` is a `WakeupOnAWTEvent`, line 76 invokes method `getAWTEvent` of class `WakeupOnAWTEvent`, which returns the array of `AWTEvent`s that triggered the `Behavior`. Line 79 passes this array to method `moveShape` (lines 89–160), which translates the user-navigated shape, depending on which key the user pressed. Lines 95–101 test if each `AWTEvent` in the array is associated with a key press—line 104 determines the specific key pressed. Lines 106–116 declare a `Transform3D` and a `Vector3f` for updating the 3D shape’s position. The `Vector3f` holds the coordinates that represent the translational component of the 3D shape’s `Transform3D`. The translational component of a 3D shape specifies the shape’s position on the x, y and z-axis. Lines 120–146 use a `switch` statement to update the 3D shape’s position according to the key the user pressed. Figure 4.26 lists the keys and corresponding translations that are valid for `Navigator`. The `Vector3f`’s x-component corresponds to the left and right position (X-axis) of a shape. The `Vector3f`’s y-component corresponds to the up and down position (Y-axis) of a shape. The `Vector3f`’s z-component corresponds to the back and forward position (Z-axis) of a shape. The `switch` statement modifies the appropriate component of the `Vector3f`. Lines 150–153 call method `setTranslation` of class `Transform3D` and method `setTransform` of class `TransformGroup` to make the translation. The Java 3D engine then updates the 3D scene with the modified `TransformGroup` information.

We have implemented `Behavior` for detecting `Node` collision (which causes the user to lose the game) and the `Behavior` for enabling the user to navigate the scene. We now implement class `GoalDetector` (Fig. 4.27)—the `Behavior` for checking the position of a 3D shape (which allows the user to win the game). Line 25 declares the `TransformGroup` for the 3D-shape’s position to check. Line 27 declares a `Switch` of scenes to display in the `SimpleUniverse`. The `SimpleUniverse` reference (line 28) adjusts the `ViewPlatform` when displaying different scenes in the `Switch`. We implement the target shape as a sphere with coordinates `goalX`, `goalY` and `goalZ` (line 30) and radius `sphereRadius` (line 33). The user wins the game when the user-navigated shape reaches the target sphere. Line 36 declares the `WakeupCondition` for the position-checking behavior.

<table>
<thead>
<tr>
<th>Key</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>move left</td>
</tr>
<tr>
<td>D</td>
<td>move right</td>
</tr>
</tbody>
</table>

*Fig. 4.26* Keys for navigating the 3D scene in *Navigator* (part 1 of 2).
Chapter 4 Graphics Programming with Java 2D and Java 3D

### Key

<table>
<thead>
<tr>
<th>Key</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>move up</td>
</tr>
<tr>
<td>S</td>
<td>move down</td>
</tr>
<tr>
<td>Up Arrow</td>
<td>move forward</td>
</tr>
<tr>
<td>Down Arrow</td>
<td>move backward</td>
</tr>
</tbody>
</table>

---

**Fig. 4.26** Keys for navigating the 3D scene in **Navigator** (part 2 of 2).

```java
// Class GoalDetector defines a position-checking behavior that
// checks to see if the position of a Node is equal to the target
// position. If the positions are equal, the game is over and
// a Java 3D Switch displays a different scene.
package com.deitel.advjhtp1.java3dgame;

// Core Java packages
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// Java extension packages
import javax.media.j3d.*;
import javax.vecmath.*;

// Java 3D utility packages
import com.sun.j3d.utils.universe.*;

public class GoalDetector extends Behavior {

    // index of scene to display if goal detected
    private static final int WINNER_SCENE = 2;

    // TransformGroup associated with object
    private TransformGroup objectTransform;

    private Switch switchScene; // Switch group that contains scenes
    private SimpleUniverse simpleUniverse;

    private float goalX, goalY, goalZ; // goal coordinates

    // radius of sphere at goal coordinates
    private float sphereRadius;

    // Behavior's waking conditions
    private WakeupCondition wakeupCondition;

    // Constructor
    public GoalDetector() {
        // Initialize variables
        // Add code for initializing the GoalDetector behavior
    }

    // Handle events
    @Override
    public void processInputEvent(InputEvent iEvent) {
        // Handle input events
        // Add code for processing input events
    }

    // Wake up the behavior if the position is equal to the goal position
    @Override
    public void update() {
        // Update behavior
        // Add code for updating the GoalDetector behavior
    }

    // Display the winning scene
    @Override
    public void execute() {
        // Execute behavior
        // Add code for displaying the winning scene
    }
}
```

---

**Fig. 4.27** Implementing a position-checking **Behavior** (part 1 of 5).
public GoalDetector( SimpleUniverse universe, TransformGroup transform, Switch switchGroup, Vector3f goalVector, float radius )
{
    objectTransform = transform;
    switchScene = switchGroup;
    simpleUniverse = universe;

    // set goal coordinates to goalVector coordinates
    goalX = goalVector.x;
    goalY = goalVector.y;
    goalZ = goalVector.z;

    // set radius of sphere at goal coordinates
    sphereRadius = radius;

    // initialize WakeupOnAWTEvent to respond to
    // AWT KeyEvent.KEY_PRESSED events
    WakeupOnAWTEvent wakeupEvent =
        new WakeupOnAWTEvent( KeyEvent.KEY_PRESSED );

    // set of WakeupEvents to which Behavior responds
    WakeupCriterion[] wakeupArray = { wakeupEvent };

    // Behavior responds when WakeupEvent in
    // WakeupCriterion occurs
    wakeupCondition = new WakeupOr( wakeupArray );
}

public void initialize()
{
    // register WakeupCriterion to respond to AWTEvents
    wakeupOn( wakeupCondition );
}

public void processStimulus( Enumeration detected )
{
    // loop to handle events
    while ( detected.hasMoreElements() ) {
        // get next sequential WakeupCriterion
        WakeupCriterion wakeupCriterion =
            ( WakeupCriterion ) detected.nextElement();

        // handle if WakeupOnAWTEvent
        if ( wakeupCriterion instanceof WakeupOnAWTEvent ) {

Fig. 4.27 Implementing a position-checking Behavior (part 2 of 5).
90  // ensure WakeupOnAWTEvent is KeyEvent.KEY_PRESSED
91  WakeupOnAWTEvent awtEvent =
92      ( WakeupOnAWTEvent ) wakeupCriterion;
93  AWTEvent[] event = awtEvent.getAWTEvent();
94
95  // check object position
96  checkPosition( event );
97
98  // re-register WakeupCriterion to respond to next
99  // key press
100  wakeupOn( wakeupCondition );
101 }
102 }
103
104 } // end method processStimulus
105
106 // check position of object in objectTransform TransformGroup
107 private void checkPosition( AWTEvent[] awtEvents )
108 {
109     Vector3f translate = new Vector3f();
110     Transform3D transform3d = new Transform3D();
111
112     // get Transform3D associated with objectTransform
113     objectTransform.getTransform( transform3d );
114
115     // get Transform3D's translation vector
116     transform3d.get( translate );
117
118     // handle all key presses in awtEvents
119     for ( int x = 0; x < awtEvents.length; x++ ) {
120
121         // handle if AWTEvent is KeyEvent
122         if ( awtEvents[ x ] instanceof KeyEvent ) {
123             KeyEvent keyEvent = (KeyEvent) awtEvents[ x ];
124
125             // handle if KeyEvent.KEY_PRESSED
126             if ( keyEvent.getID() == KeyEvent.KEY_PRESSED ) {
127
128                 // if object position == goal coordinates
129                 if ( atGoal( translate ) ) {
130                     Transform3D shiftBack = new Transform3D();
131
132                     // set translation to 8.0 on +z-axis
133                     shiftBack.setTranslation(
134                         new Vector3f( 0.0f, 0.0f, 8.0f ) );
135
136                     // set Transform3D that determines view
137                     // in SimpleUniverse
138                     ViewingPlatform viewPlatform =
139                         simpleUniverse.getViewingPlatform();
140
141                     TransformGroup platformTransform =
142                         viewPlatform.getViewPlatformTransform();
143
144                     // set Transform3D to view platform
145                     platformTransform.setTransform( shiftBack );
146
147                     // set Transform3D to determine view
148                     simpleUniverse.setTransform3D( shiftBack );
149
150                     // re-register WakeupCriterion to respond to next
151                     // key press
152                     wakeupOn( wakeupCondition );
153                 }
154             }
155         }
156     }
157 }

Fig. 4.27  Implementing a position-checking **Behavior** (part 3 of 5).
platformTransform.setTransform( shiftBack );

// render winner scene in SimpleUniverse
switchScene.setWhichChild( WINNER_SCENE );

// end if KeyEvent

// end for loop that handles key presses

} // end method checkPosition

// helper method returns true if current position is within
goal boundary
private boolean atGoal( Vector3f currentPosition )
{
    // calculate difference between current location and goal
    float x = Math.abs( currentPosition.x - goalX );
    float y = Math.abs( currentPosition.y - goalY );
    float z = Math.abs( currentPosition.z - goalZ );

    // return true if current position within sphereRadius of
goal coordinates
    return ( x < sphereRadius ) && ( y < sphereRadius ) &&
            ( z < sphereRadius );
}
The constructor method (lines 40–68) takes five arguments. The first three arguments are the `SimpleUniverse`, `TransformGroup` and `Switch`. The fourth argument is a `Vector3f` that contains the target shape’s location. Lines 49–51 set the target-point coordinates by extracting the x, y and z coordinates from the `Vector3f`. The final argument is the target-sphere radius. Lines 58–59 create a `WakeupOnAWTEvent` that responds when the user presses a key. `GoalDetector` then checks the user-navigated shape’s position to see if that shape has reached the target sphere. Line 62 creates a `WakeupCriterion` from the `WakeupOnAWTEvent`, and line 66 creates a `WakeupOr` from the `WakeupCriterion`. Method `initialize` (lines 71–75) registers the `WakeupOr` with the behavior scheduler by calling method `wakeupOn` of superclass `Behavior`.

Method `processStimulus` (lines 78–104) handles the logic for the triggered `Behavior`. This method takes as an argument an `Enumeration` of the `WakeupCriterion` that generated the behavioral event. Lines 84–85 retrieve each `WakeupCriterion` from the `Enumeration`. If the `WakeupCriterion` is a `WakeupOnAWTEvent`, line 93 invokes method `getAWTEvent` of class `WakeupOnAWTEvent`, which returns the array of the `AWTEvent`s that triggered the `Behavior`. Line 96 passes this array to method `checkPosition` (lines 107–155), which implements the position-checking algorithm.

Method `checkPosition` (lines 107–155) checks the position of the 3D shape. This method determines the 3D shape’s position by checking the `Vector3f`, which represents that shape’s translational component. Line 113 calls method `getTransform` of class `TransformGroup` to retrieve the `Transform3D` associated with the `TransformGroup`. Line 116 calls method `get` of class `Transform3D`, which retrieves the `Vector3f` that represents the translational component of the 3D shape. Lines 119–126 check if each `AWTEvent` in the array is a `KeyEvent.KEY_PRESSED` event—line 129 then calls method `atGoal`, which returns a `boolean` variable that represents whether the

Fig. 4.27 Implementing a position-checking `Behavior` (part 5 of 5).
user-navigated shape has reached the target shape. Method `atGoal` (lines 159–170) takes as an argument the `Vector3f` that contains the coordinates of the user-navigated shape’s current position. Lines 162–164 determine the absolute difference between the target shape’s coordinates (`goalX`, `goalY` and `goalZ`) and the user-navigated shape’s current coordinates. Lines 168–169 return `true` if the absolute difference for each coordinate is within the `sphereRadius`, indicating that the shape has reached the target sphere. If method `atGoal` returns `true`, the user has won the game. Lines 130–144 set the translation component of the `ViewPlatform`’s `Transform3D`; the camera shifts back to expand the view. Line 147 switches to the winning scene: rotating 3D text that reads “You Win.” Line 147 invokes method `setWhichChild` of class `Switch` passing as an argument the index of the winning scene. Figure 4.27 illustrates the game both immediately before and after the navigable object has reached the target shape.

**User Interface**

Using the `ControlPanel1` (Fig. 4.21), the user can specify the number of flying obstacles to control the game difficulty. Line 23 declares the `JSlider` that the player uses to specify the number of obstacles. Line 24 declares a `Java3DWorld1` reference through which `ControlPanel1` can set the user-specified number. The `ControlPanel1` constructor (lines 27–75) accepts as an argument a `Java3DWorld1`—line 29 sets this argument as `ControlPanel1`’s `Java3DWorld1` reference. Lines 44–50 create a `JSlider` that assumes any integer value from 1 to 20, inclusive. Lines 53–62 create a `ChangeListener` for the `JSlider`. When the player uses this `JSlider`, method `stateChanged` (lines 56–60) passes the number of obstacles to method `switchScene` of `Java3DWorld1`. The display then reveals a new scene with the specified number of obstacles.

```java
public class ControlPanel1 extends JPanel {

    private static final int CONTAINER_WIDTH = 250;
    private static final int CONTAINER_HEIGHT = 150;
    private static final int NUMBER_OF_SHAPES = 20;

    // JSliders control lighting color
    private JSlider numberSlider;
```

Fig. 4.28 Implementing Swing controls for the `Java3DWorld1` (part 1 of 3).
private Java3DWorld1 java3DWorld1;

// ControlPanel constructor
public ControlPanel1( Java3DWorld1 tempJ3DWorld )
{
    java3DWorld1 = tempJ3DWorld;

    // assemble lighting color control panel
    JPanel colorPanel = new JPanel(
        new FlowLayout( FlowLayout.LEFT, 15, 15 ) );

    TitledBorder colorBorder =
        new TitledBorder( "How Many Shapes?" );

    colorBorder.setTitleJustification( TitledBorder.CENTER );
    colorPanel.setBorder( colorBorder );

    JLabel numberLabel = new JLabel( "Number of Shapes" );

    // create JSlider for adjusting number of flying shapes
    numberSlider = new JSlider(
        SwingConstants.HORIZONTAL, 1, NUMBER_OF_SHAPES, 1 );

    numberSlider.setMajorTickSpacing( 4 );
    numberSlider.setPaintTicks( true );
    numberSlider.setPaintTrack( true );
    numberSlider.setPaintLabels( true );

    // create ChangeListener for JSliders
    ChangeListener slideListener = new ChangeListener() {
        // invoked when slider has been accessed
        public void stateChanged( ChangeEvent event )
        {
            java3DWorld1.switchScene( numberSlider.getValue(),
                NUMBER_OF_SHAPES );
        }
    };// end anonymous inner class

    // add listener to sliders
    numberSlider.addChangeListener( slideListener );

    // add lighting color control components to colorPanel
    colorPanel.add( numberLabel );
    colorPanel.add( numberSlider );
    add( colorPanel );

    // set GridLayout
    setLayout( new GridLayout( 2, 1, 0, 20 ) );

} // end ControlPanel1 constructor method

Fig. 4.28 Implementing Swing controls for the Java3DWorld1 (part 2 of 3).
This concludes our discussion of the Java 3D API. In this section, we presented a brief overview of Java 3D graphics programming. We have explained how the performance, scalability and simplicity of Java 3D make it an excellent choice for developers to incorporate 3D graphics into applications. We presented two applications that demonstrated Java 3D geometry, textures, lighting and behaviors.

We have discussed several of Java’s graphics capabilities. We began with a brief introduction to fundamental graphics topics, including coordinate systems and graphics contexts. We then discussed several Java 2D capabilities, such as controlling how to fill shapes with colors and patterns. We also introduced how to blur, invert, sharpen and change the color of an image using Java 2D’s image-processing capabilities. The second half of our graphics discussion presented the Java 3D API. Using the Java 3D utility classes, we built an application that allows the user to change properties of a Java 3D scene, including manipulating (rotate, scale and translate) 3D objects with a mouse and changing a scene’s lighting. In Chapter 5, we use Java 2D in the Deitel drawing application. We also introduce design patterns—proven strategies for creating reusable and extensible software—and use them to build this program.

SUMMARY

- A coordinate system is a scheme for identifying every point on the screen.
- The upper-left corner of a GUI component has the coordinates (0, 0).
- A graphics context enables drawing on the screen. A Graphics object manages a graphics context by controlling how information is drawn.
- Graphics objects contain methods for drawing, font manipulation, color manipulation, etc.
- Method paint is called in response to an event such as uncovering a window.
- Method repaint requests a call to method update of class Component as soon as possible to clear the Component’s background of any previous drawing. Method update then calls paint directly.
- The Swing painting mechanism calls method paintComponent of class JComponent when the contents of the JComponent should be painted.
- The Java 2D provides advanced two-dimensional graphics capabilities for processing line art, text and images.
- Class java.awt.Graphics2D enables drawing with the Java 2D API.
• To access the Graphics2D capabilities, we cast the Graphics reference passed to paint to a Graphics2D reference.

• There are seven Graphics2D attributes that determine how graphics primitives are rendered—clipping, compositing, font, paint, rendering hints, stroke and transforms.

• Method setPaint of class Graphics2D sets the Paint object that determines the color for the shape to display. A Paint object is an object of any class that implements interface java.awt.Paint. The Paint object can be a Color or an instance of the Java 2D API’s GradientPaint, SystemColor or TexturePaint classes.

• Class GradientPaint paints a shape in gradually changing colors—known as a gradient.

• Method fill of class Graphics2D draws a filled Shape object. The Shape object is an instance of any class that implements interface Shape.

• A general path is a shape constructed from lines and complex curves represented with an object of class GeneralPath (package java.awt.geom).

• Method moveTo of class GeneralPath specifies the first point in a general path. Method lineTo of class GeneralPath draws a line to the next point in the general path. Each new call to lineTo draws a line from the previous point to the current point. Method closePath of class GeneralPath draws a line from the last point to the point specified in the last call to moveTo.

• Method translate of class Graphics2D moves the drawing to a new location. All drawing operations now use that location as (0, 0).

• Image processing is the manipulation of digital images by applying filters.

• There are three main types of image-processing filters. Compression filters reduce a digital image’s memory usage. Measurement filters collect data from digital images. Enhancement filters appropriate and interpolate missing parts of corrupted images from the existing information.

• A BufferedImage separates image data into a Raster and a ColorModel. A Raster organizes and stores the numerical data that determine a pixel’s color. The ColorModel is an interpreter that takes the sample values in the Raster and converts them to different colors depending on color scale the image.

• Java 2D image-processing filters operate on objects of class BufferedImage.

• Interfaces BufferedImageOp and RasterOp serve as the base classes for Java 2D image filters. A BufferedImageOp processes a BufferedImage, while a RasterOp only processes the Raster associated with a BufferedImage.

• Method filter takes as arguments a source image and a destination image. The source image is filtered to produce the destination image.

• A LookupOp is an array indexed by source pixel color values that contains destination pixel color values.

• A sharpening filter detects edges by looking for differences in neighboring pixel sample values and enhances the edge by enlarging the difference between the sample values and is created with a ConvolveOp.

• A ConvolveOp combines the colors of a source pixel and its surrounding neighbors to determine the color of the corresponding destination pixel.

• A Kernel is a 2D array that specifies how a ConvolveOp filter should combine neighboring pixel values.

• Edge hints instruct the filter on how to alter pixels at the perimeter of the image. EDGE_NO_OP instructs the filter to copy the pixels at the source perimeter directly to the destination image without modification. EDGE_ZERO_FILL instructs the filter to fill the pixels at the perimeter of the destination with the value 0.
A blurring filter averages each pixel value with that of its eight neighboring pixels, smoothing distinct edges and is created using a `ConvolveOp`.

Each color band in a `TYPE_INT_RGB BufferedImage` is defined by three coefficients that represent the R, G and B components in the band.

We can change the colors in an image by altering the values of the R, G and B coefficients in a color band using a `BandCombineOp`. A `BandCombineOp` operates on the color bands of a `Raster`.

The Java 3D API requires that you have either OpenGL or Direct3D installed on your computer. The Java 3D API also requires you to install the appropriate Java extension and utility packages found at [java.sun.com/products/java-media/3D/download.html](http://java.sun.com/products/java-media/3D/download.html).

The root node of the Java 3D scene is a `VirtualUniverse` that has a coordinate-system, which describes the location of scene graphs.

A scene graph is a tree-like structure that contains nodes, which describe all attributes of the 3D environment. Each scene graph attaches to the `VirtualUniverse` at a specified point in the `VirtualUniverse`'s coordinate-system.

Class `Locale` is the root node in a scene graph, which contains the attachment coordinate for the `VirtualUniverse` and a number of branch graphs.

There are two types of branch graphs—content-branch graphs and view-branch graphs. View-branch graphs contain collections of objects that specify the perspective, position, orientation and scale of 3D scenes. Content-branch graphs describe the geometry, lighting, textures, fog, sound and behaviors in the 3D scenes.

Class `SceneGraphObject` is the base class for all objects in a Java 3D branch graph. `SceneGraphObject` has two subclasses—`Node` and `NodeComponent`.

Class `Group` serves as the general-purpose grouping `Node`.

`Leaf` subclasses include `Behavior`, `Light` and `Shape3D`.

`NodeComponent` objects describe the attributes of `Groups` and `Leaves`.

`Canvas3D` is a `Canvas` subclass that supports 3D rendering.

Class `SimpleUniverse` encapsulates all objects in the virtual universe and viewing platform.

Class `BranchGroup` is the root node of a scene graph.

Class `TransformGroup` specifies transformations including rotation, scaling and translation.

To modify an object in a scene in run time, the developer must set that object's capability bits using method `setCapability`.

All content `Leaves` in Java 3D are bounded by a volume that defines the space in which the `Leaves` are rendered.

Class `Appearance` describes the attributes of the 3D geometry and has associated attribute objects, such as `Material` and `Texture`.

Class `Material` defines the properties of any object that falls under illumination.

Class `com.sun.j3d.utils.image.TextureLoader` loads an `Image` for texturing geometry.

Class `AmbientLight` is a light source that illuminates all shapes evenly within its bounds.

Class `DirectionalLight` is a light source that travels from a source point to a destination point.

When a `Light` source is added to a `Group`, all objects in that `Group` are illuminated.

The `MouseBehavior` classes in utility package `com.sun.j3d.utils.behavior.mouse` help developers integrate mouse interaction into applications.

Classes `MouseRotate`, `MouseTranslate` and `MouseZoom` allow the user to use a mouse to rotate, translate and scale a 3D shape, respectively.
• Method `compile` of class `BranchGroup` causes the `BranchGroup` and all its children to be compiled.

• A `Switch` group specifies which of its children to render. A `Switch` can render either one child at a time or several children at once.

• Interpolators use `Alpha`s to specify certain characteristics of animation, such as the speed of transformations (e.g., rotation speed), or how fast a shape changes color (e.g., lighting effects).

• Interpolators operate in conjunction with a `TransformGroup`: each `TransformGroup` has an associated `Interpolator`. The `Interpolator` describes how to animate shapes in the `TransformGroup`.

• An `Alpha` object generates the time values to the `Interpolator`. `Alpha` objects consist of a series of phases that can be either increasing or decreasing.

• Class `Transform3D` is a two-dimensional array that represents a general transform. Each `Transform3D` has an associated integer type that determines the transformation to represent.

• A `Material`’s ambient color is the `Material`’s color when illuminated by reflected light. A `Material`’s diffuse color is the `Material`’s color when illuminated by some light source.

• The Java 3D `com.sun.j3d.utils.geometry` package provides four types of 3D geometric `Primitive` objects: Box, Cone, Cylinder and Sphere.

• Class `Text3D` is a `Geometry` subclass for representing three-dimensional text.

• A `Font3D` object is constructed from a `java.awt.Font` object and a Java 3D `FontExtrusion` object. A `FontExtrusion` describes process of adding a third dimension to the `Font`’s 2D text.

• A `Point3f` specifies x-y-z coordinates in a 3D `SimpleUniverse`.

• The Java 3D API provides the abstract `Behavior` class to create a variety of custom behaviors.

• A `Behavior` object has an associated behavior scheduler responsible for registering wake-up conditions.

• All classes that extend `Behavior` must implement methods `initialize` and `process-Stimulus`.

• Collision detection determines when a shape’s bounding volume—the volume enclosing either a shape or the bounds of a shape—intersects another.

• A collision occurs when an armed node’s bounding volume intersects a triggering node’s bounding volume.

• When a `WakeupCondition` is satisfied, the behavior scheduler calls method `process-Stimulus`, passing as an argument an `Enumeration` of the `WakeupCriteria`s that triggered the event.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Method/Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addChild method</td>
<td>bounding volume class</td>
</tr>
<tr>
<td><code>alpha</code> values</td>
<td><code>BoundingBox</code> class</td>
</tr>
<tr>
<td><code>AmbientLight</code> class</td>
<td><code>BoundingSphere</code> class</td>
</tr>
<tr>
<td><code>Appearance</code> class</td>
<td><code>Box</code> class</td>
</tr>
<tr>
<td>arc angle</td>
<td>branch graph</td>
</tr>
<tr>
<td><code>Arc2D.Double</code> class</td>
<td><code>BranchGroup</code> class</td>
</tr>
<tr>
<td><code>BandCombineOp</code> class</td>
<td><code>BufferedImage</code> class</td>
</tr>
<tr>
<td><code>BasicStroke</code> class</td>
<td><code>BufferedImageOp</code> interface</td>
</tr>
<tr>
<td><code>Behavior</code> class</td>
<td><code>ByteLookupTable</code> class</td>
</tr>
<tr>
<td>bounding rectangle</td>
<td><code>Canvas3D</code> class</td>
</tr>
</tbody>
</table>
capability bits
clipping
**closePath** method
**Color3f** class
collision detection
**Color** class
color bands
color scale
**ColorModel** class
compositing
compression filters
**ConvolveOp** class
coordinate system
**Component** class
**DataBuffer** class
Direct3D
**DirectionalLight** class
draw method
drawImage method
drawRect method
drawString method
edges
**Ellipse2D.Double** class
enhancement filter
event-driven process
**fill** method
**fillRect** method
**filter** method
fog
**Font** class
**GeneralPath** class
geometry
**getColorModel** method
**getPreferredConfiguration** method
**getRaster** method
**getViewingPlatform** method
gradient
**GradientPaint** class
**Graphics** class
graphics context
graphics primitives
**Graphics2D** class
grayscale
**Group** class
image processing
Java 2D API
Java 3D API
**JComponent** class
**Kernel** class
**Leaf** class
**Light** class
line joins
**Line2D.Double** class
**lineTo** method
**Locale** class
**LookupOp** class
machine vision
**Material** class
measurement filters
modeled
morphing
**MouseBehavior** class
**MouseRotate** class
**MouseZoom** class
**MouseTranslate** class
**moveTo** method
**Node** class
**NodeComponent** class
OpenGL
optimization
**Paint** interface
**paint** method
**paintComponent** method
pixel
polygon
**Raster** class
**RasterFormatException** class
rasterize
**RasterOp** interface
**Rectangle2D.Double** class
render
rendering engine
rendering hints
rendering pipeline
**RenderingHints** class
**repaint** method
RGB value
rotate
**RoundRectangle2D.Double** class
sample
**SampleModel** class
scene
scene graph
**SceneGraphObject** class
**setColor** method
**setCapability** method
**setEnable** method
**setInfluencingBounds** method
**setMaterial** method
SELF-REVIEW EXERCISES

4.1 Fill in the blanks in each of the following statements:

a) In Java 2D, class ________ defines the fill for a shape such that the fill gradually changes from one color to another.

b) In Java 2D, an image-processing filter that operates on both a pixel and its neighboring pixels is implemented using class ________.

c) Class ________ stores pixel sample data in a BufferedImage, while class ________ contains instructions for translating the pixel sample to a color.

d) Rotation, scaling and translation are all examples of ________.

e) Method ________ of class DirectionalLight sets a flag that alerts the compiler that the DirectionalLight’s attributes should be writable during execution.

f) In Java 3D, class ________ contains NodeComponents that describe the attributes of a shape, including Material and Texture.

4.2 State whether each of the following is true or false. If false, explain why.

a) The LookupOp constructor takes as arguments a Kernel and a RenderingHints object.

b) Method closePath of class GeneralPath to draw a line from the last point to the point specified in the first call to moveTo.

c) The source and destination Raster arguments to the BandCombineOp constructor can be the same Rasters.

d) In Java 3D, Behaviors do not affect objects outside the Behavior's bounding volume.

e) Class SimpleUniverse creates a Java 3D scene that contains a VirtualUniverse, Locale and view branch graph.

f) All children in a BranchGroup will be affected behaviors defined in TransformGroup objects that are part of that BranchGroup.

ANSWERS TO SELF-REVIEW EXERCISES


4.2 a) False. The arguments to the LookupOp constructor are a LookupTable that contains the color sample lookup array and a RenderingHints object. b) False. Method closePath draws a line from the last point to the point specified in the last call to moveTo. c) True. d) True. e) True. f) False. Only those Nodes that are children of the TransformGroup will be affected by the TransformGroup’s behavior. Any Nodes outside the TransformGroup are not affected by the TransformGroup’s behavior.
EXERCISES

4.3 Write a program that draws a pyramid. Use class `GeneralPath` and method `draw` of class `Graphics2D`.

4.4 Write a program that draws a series of eight concentric circles that are separated by 10 pixels using class `Ellipse2D.Double`. The outer seven circles should be filled with randomly generated solid colors. The innermost circle should be filled with a gradient. Use method `draw` of class `Graphics2D`.

4.5 Modify the image-processing program presented in this chapter to include an `Java2DImageFilter` that removes the green color band from a `BufferedImage`. Add this option to the menu created in Fig. 4.13.

4.6 Modify the program of Fig. 4.15 and Fig. 4.21 so that the set of `JSlider` controls affect the direction of the `DirectionalLight` source as opposed to the color. Use method `setDirection` of class `DirectionalLight`.

4.7 For the program of Fig. 4.23, create a `Behavior` that temporarily “shields” the user-navigated sphere from a collision with an obstacle. When the user presses the space-bar, the user-navigated sphere turns blue to indicate that it is “shielded”—the sphere is “invincible” and should not collide with any obstacles for three seconds (i.e., during this time, the sphere can pass through obstacles without the “You Lose” screen appearing). The user can use the “shield” feature three times per game—after that, pressing the space-bar has no effect. When the shield “wears off,” the user-navigated shape should turn red to indicate that it can collide with obstacles.
Case Study: Java 2D GUI Application with Design Patterns

Objectives

- To understand the model-view-controller architecture in a GUI application.
- To understand drag-and-drop techniques for transferring data in and among applications.
- To understand the Factory Method design pattern for creating objects based on runtime criteria.
- To understand the integration of multiple Java technologies to build applications.
- To understand the use of multiple design patterns in a single application.
- To understand the implementation of multiple-document-interface applications.

All my life I have struggled to make one authentic gesture.
Isadora Duncan

Whatever is in any way beautiful has its source of beauty in itself, and is complete in itself; praise forms no part of it.
Marcus Aurelius Antoninus

The source of genius is imagination alone, ...the refinement of the senses that sees what others do not see, or sees them differently.
Eugene Delacroix

That is a transformation in which imagination collaborates with memory.
Edgar Degas
5.1 Introduction

In this chapter, we implement a Java application case study as a capstone for the many Java features and techniques presented in previous chapters, including Swing GUI components and Java 2D graphics. This case study is a substantial application with almost 4,000 lines of code, so we use several design patterns to facilitate proper object-oriented design and extensibility. These design patterns include some we introduced in previous chapters (e.g., the Command design pattern and the MVC architecture), and some that we introduce in this case study.

5.2 Application Overview

The Deitel Drawing application is a painting program that enables users to create drawings that contain lines, shapes, text and images. Deitel Drawing includes the following features:

1. Colors, filled shapes and gradients
2. Multiple-document interface
3. Drag-and-drop support for moving shapes between drawings
4. Drag-and-drop support for JPEG images
5. Saving drawings as XML documents
6. Scaling drawings to different sizes and aspect ratios
7. Multiple drawing tools (controllers)
8. Modifying shape properties such as line width, fill and gradient

Deitel Drawing uses the model-view-controller architecture to make the application modular and extensible. The model consists of a collection of objects that extend abstract
base class `MyShape`. Using polymorphism, views create graphical presentations of the `MyShape` collections. Multiple controllers handle input for drawing `MyShape` subclasses and for processing drag-and-drop operations.

Deitel Drawing uses the Java2D graphics APIs to create high-quality graphical presentations of drawing models. Lines, shapes and fonts are drawn using anti-aliasing to smooth jagged edges. Deitel Drawing takes advantage of Java 2D's `GradientPaint` class to draw shapes using multicolor gradients. Java 2D also provides transformation capabilities that enable the application to display scaled views of drawings. Using Java's event-handling mechanism, Deitel Drawing allows users to scale drawings dynamically by resizing a `ZoomDialog` window. The model-view-controller architecture ensures that each view is consistent with the drawing stored in the model. As a user draws new shapes, those shapes are immediately shown in each view.

Enabling drag-and-drop functionality in applications is nontrivial. The Deitel Drawing application uses Java’s sophisticated drag-and-drop API to implement drag-and-drop functionality that allows users to move objects between drawings easily. Users also can drag and drop JPEG images from other applications (such as the host operating system’s file manager) into drawings. Once the JPEG image is part of the drawing, the user can drag and drop the image between drawings just as with other shapes.

Figure 5.1 shows the Deitel Drawing application with a sample drawing. The shapes in this drawing were generated randomly by the solution to Exercise 5.8. Figure 5.2 shows the same drawing scaled to approximately twice the original size in a `ZoomDialog`.

### 5.3 MyShape Class Hierarchy

Deitel Drawing represents each shape in a drawing as a separate object that extends class `MyShape`. `MyShape` is an abstract base class that defines the basic interface for shapes and default implementations for methods common to all shapes.

Class `MyShape` (Fig. 5.3) is the root of the shape-class hierarchy. Implementing interface `Serializable` enables the Deitel Drawing application to serialize `MyShape` objects to disk, so drawings can be saved.

Lines 17–24 define several properties common to all `MyShapes`, such as the x- and y-coordinates and the `MyShape`'s colors. Some `MyShapes` can be filled (e.g., a filled square) or drawn with a gradient (lines 20–21). Line 22 declares property `strokeSize`, which specifies the thickness of the shape’s lines. Methods `getLeftX` (lines 26–29) and `getLeftY` (lines 32–35) return the x- and y-coordinates of the `MyShape`'s left-most point. Methods `getRightX` (lines 38–41) and `getRightY` (lines 44–47) return the x- and y-coordinates of the `MyShape`'s right-most point. Methods `getWidth` (lines 50–53) and `getHeight` (lines 56–59) return the `MyShape`'s width and height as calculated from the shape’s coordinates. Methods `setPoint1` (lines 62–66) and `setPoint2` (lines 69–73) modify the shape’s x- and y-coordinates. Methods `setStartPoint` (lines 76–80) and `setEndPoint` (lines 83–87) set the points at which drawing began and drawing ended. The `MyShape` uses the start and end points to determine how to draw its gradient. Lines 90–136 provide get methods for each individual x- and y-coordinate. Method `moveByOffSet` (lines 139–145) moves the `MyShape` by the given x and y offset values.
Lines 148–181 provide set and get methods for each of the MyShape’s colors. MyShapes can be drawn either in their primary color (startColor) or with a gradient that starts with startColor and ends with endColor. Lines 184–193 provide set and get methods for the useGradient property, which, if true, draws the shape using a color gradient.

Lines 196–206 provide set and get methods for property strokeSize. The Java2D API uses strokes to draw objects on a graphics context. The strokeSize property determines the thickness of the line that strokes the shape. Lines 209–218 provide set and get methods for property filled, which specifies whether the shape should be filled or drawn as an outline.

Line 222 declares abstract method draw, which takes as a Graphics2D argument the graphics context on which to draw the shape. Method draw is abstract because a generic MyShape object cannot be drawn; only specific subclasses of class MyShape (e.g., MyOval) can be drawn. Method contains (line 225) returns true if the given Point2D falls within the MyShape’s area. Method contains also is declared abstract to require each subclass to define an appropriate implementation. The drag-and-drop implementation in this example uses method contains when beginning a drag operation.

Method configureGraphicsContext (lines 228–247) configures the given Graphics2D object for drawing this MyShape. If there does not exist a Stroke for drawing the shape, line 233 creates a BasicStroke object using MyShape’s strokeSize property. Line 234 sets the Graphics2D object’s stroke property. If the gradient property is true, lines 239–242 create a GradientPaint object that begins with startColor and ends with endColor. The gradient extends from the point (startX, startY) to the point (endX, endY). If the gradient property is false, line 246 invokes method setPaint of class Graphics2D to use the MyShape’s default Color.


Class MyLine (Fig. 5.4) is a MyShape subclass that represents a line in the drawing. Lines 15–26 implement method draw, which was declared abstract in class MyShape. Line 18 invokes method configureGraphicsContext to configure the given Graphics2D object with the MyLine object’s color, strokeLineSize and other properties. Lines 21–22 create a Java2D Line2D.Float object for the MyLine object’s x- and y-coordinates. Class Line2D.Float represents a line using floats for its x- and y-coordinates. Line 25 invokes method draw of class Graphics2D to draw the line on the Graphics2D context.

Method contains (lines 29–32) calculates the line’s slope to determine if the given Point2D is on the line. Method getXML (lines 49–55) invokes method getXML of class MyShape (line 37) to get the default shape Element. Line 38 sets Attribute type of Element shape to the value MyLine to indicate that this MyShape object is an instance of class MyLine.
Fig. 5.1  Deitel Drawing application showing randomly drawn shapes (Exercise 5.8) and a ZoomDrawingView (Fig. 5.13).

Fig. 5.2  Large-scale view of drawing from Fig. 5.1.
// MyShape.java
// MyShape is an abstract base class that represents a shape
to be drawn in the DeitelDrawing application.
package com.deitel.advjhtp1.drawing.model.shapes;

// Java core packages
import java.awt.*;
import java.awt.geom.Point2D;

// third-party packages
import org.w3c.dom.*;

public abstract class MyShape {

    // MyShape properties (coordinates, colors, etc.)
    private int x1, y1, x2, y2;
    private int startX, startY, endX, endY;
    private Color startColor = Color.black;
    private Color endColor = Color.white;
    private boolean filled = false;
    private boolean gradient = false;
    private float strokeSize = 1.0f;
    private Stroke currentStroke;

    // get x coordinate of left corner
    public int getLeftX()
    {
        return x1;
    }

    // get y coordinate of left corner
    public int getLeftY()
    {
        return y1;
    }

    // get x coordinate of right corner
    public int getRightX()
    {
        return x2;
    }

    // get y coordinate of right corner
    public int getRightY()
    {
        return y2;
    }

    // get MyShape width
    public int getWidth()
    {
        return Math.abs( getX1() - getX2() );
    }

Fig. 5.3  MyShape abstract base class for drawing objects (part 1 of 7).
// get MyShape height
public int getHeight()
{
    return Math.abs( getY1() - getY2() );
}

// set Point1's x and y coordinates
public void setPoint1( int x, int y )
{
    x1 = x;
    y1 = y;
}

// set Point2's x and y coordinates
public final void setPoint2( int x, int y )
{
    x2 = x;
    y2 = y;
}

// set start Point's x and y coordinates
public final void setStartPoint( int x, int y )
{
    startX = x;
    startY = y;
}

// set end Point's x and y coordinates
public final void setEndPoint( int x, int y )
{
    endX = x;
    endY = y;
}

// get x1 coordinate
public final int getX1()
{
    return x1;
}

// get x2 coordinate
public final int getX2()
{
    return x2;
}

// get y1 coordinate
public final int getY1()
{
    return y1;
}

Fig. 5.3  MyShape abstract base class for drawing objects (part 2 of 7).
// get y2 coordinate
public final int getY2()
{
    return y2;
}

// get startX coordinate
public final int getStartX()
{
    return startX;
}

// get startY coordinate
public final int getStartY()
{
    return startY;
}

// get endX coordinate
public final int getEndX()
{
    return endX;
}

// get endY coordinate
public final int getEndY()
{
    return endY;
}

// move MyShape by given offset
public void moveByOffSet( int x, int y )
{
    setPoint1( getX1() + x, getY1() + y );
    setPoint2( getX2() + x, getY2() + y );
    setStartPoint( getStartX() + x, getStartY() + y );
    setEndPoint( getEndX() + x, getEndY() + y );
}

// set default drawing color
public void setColor( Color color )
{
    setStartColor( color );
}

// get default drawing color
public Color getColor()
{
    return getStartColor();
}

// set primary drawing color

Fig. 5.3 MyShape abstract base class for drawing objects (part 3 of 7).
```java
public void setStartColor( Color color )
{
    startColor = color;
}

public Color getStartColor()
{
    return startColor;
}

public void setEndColor( Color color )
{
    endColor = color;
}

public Color getEndColor()
{
    return endColor;
}

public void setUseGradient( boolean useGradient )
{
    gradient = useGradient;
}

public boolean useGradient()
{
    return gradient;
}

public void setStrokeSize( float size )
{
    strokeSize = size;
    currentStroke = new BasicStroke( strokeSize );
}

public float getStrokeSize()
{
    return strokeSize;
}

public void setFilled ( boolean fill )
{
    filled = fill;
}
```

---

**MyShape abstract** base class for drawing objects (part 4 of 7).
213  // get filled property
214  public boolean isFilled()
215  {
216      return filled;
217  }
218
219  // abstract draw method to be implemented by subclasses
220  // to draw actual shapes
221  public abstract void draw( Graphics2D g2D );
222
223  // return true if the Point2D falls within this shape
224  public abstract boolean contains( Point2D point );
225
226  // configure Graphics2D context for known drawing properties
227  protected void configureGraphicsContext( Graphics2D g2D )
228  {
229      // set Stroke for drawing shape
230      if ( currentStroke == null )
231          currentStroke = new BasicStroke( getStrokeSize() );
232
233          g2D.setStroke( currentStroke );
234
235      // if gradient selected, create new GradientPaint starting
236      // at x1, y1 with color1 and ending at x2, y2 with color2
237      if ( useGradient() )
238          g2D.setPaint ( new GradientPaint(  
239              ( int ) getStartX(), ( int ) getStartY(),  
240              getStartColor(), ( int ) getEndX(), ( int ) getEndY(),  
241              getEndColor() ) );
242
243      // if no gradient selected, use primary color
244      else
245          g2D.setPaint( getColor() );
246  }
247
248  // get MyShape XML representation
249  public Element getXML( Document document )
250  {
251      Element shapeElement = document.createElement( "shape" );
252
253      // create Elements for x and y coordinates
254      Element temp = document.createElement( "x1" );
255      temp.appendChild( document.createTextNode(  
256          String.valueOf( getX1() ) ) );
257      shapeElement.appendChild( temp );
258
259      temp = document.createElement( "y1" );
260      temp.appendChild( document.createTextNode(  
261          String.valueOf( getY1() ) ) );
262      shapeElement.appendChild( temp );
263
264      temp = document.createElement( "x2" );
265      temp.appendChild( document.createTextNode(  
266          String.valueOf( getX2() ) ) );
267      shapeElement.appendChild( temp );
268      
269      return shapeElement;
270  }

Fig. 5.3  MyShape abstract base class for drawing objects (part 5 of 7).
temp.appendChild( document.createTextNode( String.valueOf( getX2() ) ) );
shapeElement.appendChild( temp );

temp = document.createElement( "y2" );
temp.appendChild( document.createTextNode( String.valueOf( getY2() ) ) );
shapeElement.appendChild( temp );

temp = document.createElement( "startX" );
temp.appendChild( document.createTextNode( String.valueOf( getStartX() ) ) );
shapeElement.appendChild( temp );

temp = document.createElement( "startY" );
temp.appendChild( document.createTextNode( String.valueOf( getStartY() ) ) );
shapeElement.appendChild( temp );

temp = document.createElement( "endX" );
temp.appendChild( document.createTextNode( String.valueOf( getEndX() ) ) );
shapeElement.appendChild( temp );

temp = document.createElement( "endY" );
temp.appendChild( document.createTextNode( String.valueOf( getEndY() ) ) );
shapeElement.appendChild( temp );

// create Element for gradient property
temp = document.createElement( "useGradient" );
temp.appendChild( document.createTextNode( String.valueOf( useGradient() ) ) );
shapeElement.appendChild( temp );

// create XML element for startColor
Color color = getStartColor();
temp = document.createElement( "startColor" );
temp.setAttribute( "red", String.valueOf( color.getRed() ) );
temp.setAttribute( "green", String.valueOf( color.getGreen() ) );
temp.setAttribute( "blue", String.valueOf( color.getBlue() ) );
shapeElement.appendChild( temp );

// create XML element for endColor
color = getEndColor();
temp = document.createElement( "endColor" );
temp.setAttribute( "red", String.valueOf( color.getRed() ) );
temp.setAttribute( "green", String.valueOf( color.getGreen() ) );
shapeElement.appendChild( temp );

Fig. 5.3 MyShape abstract base class for drawing objects (part 6 of 7).
temp.setAttribute( "blue", 
    String.valueOf( color.getBlue() ) );
shapeElement.appendChild( temp );

// add strokeSize element
temp = document.createElement( "strokeSize" );
temp.appendChild( document.createTextNode( 
    String.valueOf( getStrokeSize() ) );
shapeElement.appendChild( temp );

// add fill element
temp = document.createElement( "fill" );
temp.appendChild( document.createTextNode( 
    String.valueOf( isFilled() ) );
shapeElement.appendChild( temp );

return shapeElement;
}

Fig. 5.3  MyShape abstract base class for drawing objects (part 7 of 7).

// MyLine.java
// MyLine is a MyShape subclass that represents a line.
package com.deitel.advjhtp1.drawing.model.shapes;

// Java core packages
import java.awt.*;
import java.awt.geom.*;

// third-party packages
import org.w3c.dom.*;

public class MyLine extends MyShape {

    // draw MyLine object on given Graphics2D context
    public void draw( Graphics2D g2D )
    {
        // configure Graphics2D (gradient, color, etc.)
        configureGraphicsContext( g2D );

        // create new Line2D.Float
        Shape line = new Line2D.Float( getX1(), getY1(), getX2(),
        getY2() );

        // draw shape
        g2D.draw( line );
    }

Fig. 5.4  MyLine subclass of class MyShape that represents a line (part 1 of 2).
// determine if MyLine contains given Point2D
public boolean contains( Point2D point )
{
    // get Point1 and Point2 coordinates
    float x1 = getX1();
    float x2 = getX2();
    float y1 = getY1();
    float y2 = getY2();

    // determines slope of line
    float slope = ( y2 - y1 ) / ( x2 - x1 );

    // determines slope from point argument and Point1
    float realSlope = ( float )
        ( ( point.getY() - y1 ) / ( point.getX() - x1 ) );

    // return true if slope and realSlope are close in value
    return Math.abs( realSlope - slope ) < 0.1;
}

// get MyLine XML representation
public Element getXML( Document document )
{
    Element shapeElement = super.getXML( document );
    shapeElement.setAttribute( "type", "MyLine" );

    return shapeElement;
}

Fig. 5.4 MyLine subclass of class MyShape that represents a line (part 2 of 2).

Class MyRectangle (Fig. 5.5) is a MyShape subclass that represents a rectangle.
Lines 17–31 implement method draw, which takes as a Graphics2D argument the
graphics context on which to draw the **MyRectangle**. Line 20 invokes method `configureGraphicsContext` to set the appropriate `strokeSize`, color and other drawing properties. Lines 23–24 create a new `Rectangle2D.Float` instance. The `Rectangle2D.Float` constructor takes as arguments the x- and y-coordinates of the rectangle’s upper left hand corner and the rectangle’s width and height. If `MyRectangle`’s `filled` property is set, line 28 draws a filled rectangle by invoking method `fill` of class `Graphics2D`. If `MyRectangle`’s `filled` property is false, line 30 invokes method `draw` of class `Graphics2D` to draw the rectangle’s outline.

```java
1 // MyRectangle.java
2 // MyRectangle is a MyShape subclass that represents a
3 // rectangle, including an implementation of the draw method
4 // for drawing the rectangle on a Graphics2D context.
5 package com.deitel.advjhtp1.drawing.model.shapes;
6
7 // Java core packages
8 import java.awt.*;
9 import java.awt.geom.*;
10
11 // third-party packages
12 import org.w3c.dom.*;
13
14 public class MyRectangle extends MyShape {
15     // draw MyRectangle on given Graphics2D context
16     public void draw( Graphics2D g2D )
17     {
18         // configure Graphics2D (gradient, color, etc.)
19         configureGraphicsContext( g2D );
20
21         // create Rectangle2D for drawing MyRectangle
22         Shape shape = new Rectangle2D.Float( getLeftX(),
23             getLeftY(), getWidth(), getHeight() );
24
25         // if shape is filled, draw filled shape
26         if ( isFilled() )
27             g2D.fill( shape );
28         else
29             g2D.draw( shape );
30     }
31
32     // return true if point falls within MyRectangle
33     public boolean contains( Point2D point )
34     {
35         Rectangle2D.Float rectangle = new Rectangle2D.Float(  
36             getLeftX(), getLeftY(), getWidth(), getHeight() );
37
38         return rectangle.contains( point );
39     }
40 }
```

**Fig. 5.5** **MyRectangle** subclass of class **MyShape** that represents a rectangle (part 1 of 2).
Method `contains` (lines 34–40) creates a `Rectangle2D.Float` object (line 36–37) and invokes method `contains` of class `Rectangle2D.Float` to determine whether the given `Point2D` falls within the `MyRectangle`. Method `getXML` (lines 43–49) creates an XML `Element` to represent the `MyRectangle` object. Line 45 invokes method `getXML` of class `MyShape` to get the default `shape Element`. Line 46 invokes method `setAttribute` of interface `Element` to add `Attribute type` with the value `MyRectangle` to `Element shape`.

Class `MyOval` (Fig. 5.6) is a `MyShape` subclass that represents an oval. Lines 15–29 implement method `draw` for drawing the `MyOval` object on the given `Graphics2D` context. Line 18 invokes method `configureGraphicsContext` to set the color, `strokeSize` and other properties for drawing the `MyOval`. Lines 21–22 create an `Ellipse2D.Float` instance for drawing the `MyOval`. The `Ellipse2D.Float` constructor takes as arguments the `x`- and `y`-coordinates and the width and height of the oval’s bounding rectangle. If the `MyOval` is `filled`, line 26 invokes method `fill` of class `Graphics2D` to draw a filled oval. If the `MyOval` is not `filled`, line 28 invokes method `draw` of class `Graphics2D` to draw the oval’s outline.
Method `contains` (lines 32–38) creates an `Ellipse2D.Float` object and invokes method `contains` of class `Ellipse2D.Float` to determine whether the given `Point2D` falls within the oval. Method `getXML` (lines 41–47) creates an XML `Element` to represent the `MyOval` object. Line 43 invokes method `getXML` of class `MyShape` to get the default `shape Element`. Line 44 invokes method `setAttribute` of interface `Element` to add `Attribute type` with value `MyOval` to `Element shape`.

Class `MyText` (Fig. 5.7) is a `MyShape` subclass that represents styled text in a drawing. A `MyText` object contains a `String` of text (line 18), in a particular font (line 20), of a particular size (line 21) that optionally may be bold, underlined and/or italic (lines 22–24). Method `draw` (lines 27–66) draws the `MyText` object using a `java.textAttributedString`. An `AttributedString` contains text and attributes of that text, such as its font. Line 30 invokes method `configureGraphicsContext` to initialize the `Graphics2D` object for drawing the `MyText` object. Line 33 creates an `AttributedString`, and lines 36–58 set that `AttributedString`’s attributes, including the font, size, bold, italic, etc. Line 65 invokes method `drawString` of class `Graphics2D` to draw the `AttributedString` on the graphics context. Method `contains` (lines 69–72) always returns `false`, which disallows dragging of `MyText` objects.

```java
// MyOval.java
// MyOval is a MyShape subclass that represents an oval.
package com.deitel.advjhtp1.drawing.model.shapes;

// Java core packages
import java.awt.*;
import java.awt.geom.*;

// third-party packages
import org.w3c.dom.*;

public class MyOval extends MyShape {

    // draw MyOval on given Graphics2D context
    public void draw( Graphics2D g2D )
    {
        // configure Graphics2D (gradient, color, etc.)
        configureGraphicsContext( g2D );

        // create Ellipse2D for drawing oval
        Shape shape = new Ellipse2D.Float( getLeftX(),
                                            getLeftY(), getWidth(), getHeight() );

        // if shape is filled, draw filled shape
        if ( isFilled() )
            g2D.fill( shape );
        else
            g2D.draw( shape );
    }
}
```

**Fig. 5.6** `MyOval` subclass of class `MyShape` that represents an oval (part 1 of 2).
// return true if point falls inside MyOval
public boolean contains( Point2D point )
{
    Ellipse2D.Float ellipse = new Ellipse2D.Float(
        getLeftX(), getLeftY(), getWidth(), getHeight() );

    return ellipse.contains( point );
}

// get MyOval XML representation
public Element getXML( Document document )
{
    Element shapeElement = super.getXML( document );
    shapeElement.setAttribute( "type", "MyOval" );

    return shapeElement;
}

Fig. 5.6  MyOval subclass of class MyShape that represents an oval (part 2 of 2).

// MyText.java
// MyText is a MyShape subclass that represents styled text
// in a drawing.
package com.deitel.advjhtp1.drawing.model.shapes;

// Java core packages
import java.awt.*;
import java.text.*;

Fig. 5.7  MyText subclass of class MyShape that represents a string of text (part 1 of 5).
import java.awt.font.*;
import java.awt.geom.*;

// third-party packages
import org.w3c.dom.*;

public class MyText extends MyShape {
    // MyText properties (font, font size, text, etc.)
    private String text;
    private AttributedString attributedString;
    private String fontName = "Serif";
    private int fontSize = 12;
    private boolean underlined = false;
    private boolean boldSelected = false;
    private boolean italicSelected = false;

    // draw MyText on given Graphics2D context
    public void draw( Graphics2D g2D )
    {
        // configure Graphics2D (gradient, color, etc.)
        configureGraphicsContext( g2D );

        // create AttributedString for drawing text
        attributedString = new AttributedString( text );

        // set AttributedString Font
        attributedString.addAttribute( TextAttribute.FAMILY, fontName );

        // set AttributedString Font size
        attributedString.addAttribute( TextAttribute.SIZE, new Float( fontSize ) );

        // if selected, set bold, italic and underlined
        if ( boldSelected )
            attributedString.addAttribute( TextAttribute.WEIGHT, TextAttribute.WEIGHT_BOLD );
        if ( italicSelected )
            attributedString.addAttribute( TextAttribute.POSTURE, TextAttribute.POSTURE_OBLIQUE );
        if ( underlined )
            attributedString.addAttribute( TextAttribute.UNDERLINE, TextAttribute.UNDERLINE_ON );

        // set AttributedString Color
        attributedString.addAttribute( TextAttribute.FOREGROUND, getColor() );

        // create AttributedCharacterIterator for AttributedString
    }

Fig. 5.7 MyText subclass of class MyShape that represents a string of text (part 2 of 5).
AttributedCharacterIterator characterIterator = attributedString.getIterator();

// draw string using AttributedCharacterIterator
g2D.drawString( characterIterator, getX1(), getY1() );

// return false because MyText objects contain no area
public boolean contains( Point2D point )
{
    return false;
}

// set MyText text
public void setText( String myText )
{
    text = myText;
}

// get text contained in MyText
public String getText()
{
    return text;
}

// set MyText Font size
public void setFontSize( int size )
{
    fontSize = size;
}

// get MyText Font size
public int getFontSize()
{
    return fontSize;
}

// set MyText Font name
public void setFontName( String name )
{
    fontName = name;
}

// get MyText Font name
public String getFontName()
{
    return fontName;
}

// set MyText underlined property
public void setUnderlineSelected( boolean textUnderlined )
{

Fig. 5.7 MyText subclass of class MyShape that represents a string of text (part 3 of 5).
underlined = textUnderlined;
}

// get MyText underlined property
public boolean isUnderlineSelected()
{
    return underlined;
}

// set MyText bold property
public void setBoldSelected( boolean textBold )
{
    boldSelected = textBold;
}

// get MyText bold property
public boolean isBoldSelected()
{
    return boldSelected;
}

// set MyText italic property
public void setItalicSelected( boolean textItalic )
{
    italicSelected = textItalic;
}

// get MyText italic property
public boolean isItalicSelected()
{
    return italicSelected;
}

// get MyText XML representation
public Element getXML( Document document )
{
    Element shapeElement = super.getXML( document );
    shapeElement.setAttribute( "type", "MyText" );
    // create text Element
    Element temp = document.createElement( "text" );
    temp.appendChild( document.createTextNode( getText() ) );
    shapeElement.appendChild( temp );
    // create fontSize Element
    temp = document.createElement( "fontSize" );
    temp.appendChild( document.createTextNode( String.valueOf( fontSize ) ) );
    shapeElement.appendChild( temp );
    // create fontName Element
    temp = document.createElement( "fontName" );
    MyText subclass of class MyShape that represents a string of text (part 4 of 5).
temp.appendChild( document.createTextNode(String.valueOf( fontName ) ) );
shapeElement.appendChild( temp );

// create underlined Element
temp = document.createElement( "underlined" );
temp.appendChild( document.createTextNode(String.valueOf( underlined ) ) );
shapeElement.appendChild( temp );

// create bold Element
temp = document.createElement( "bold" );
temp.appendChild( document.createTextNode(String.valueOf( boldSelected ) ) );
shapeElement.appendChild( temp );

// create italic Element
temp = document.createElement( "italic" );
temp.appendChild( document.createTextNode(String.valueOf( italicSelected ) ) );
shapeElement.appendChild( temp );

return shapeElement;
}

Fig. 5.7  MyText subclass of class MyShape that represents a string of text (part 5 of 5).
Lines 75–144 provide set and get methods for MyText properties, including its text, font, size, bold, italic and underline properties. Method getXML (lines 147–189) creates an XML representation of a MyText object. Lines 149–150 obtain the default shape Element and set its type attribute to the value "MyText". Lines 153–185 create Elements that represent each MyText-specific property.

Class MyImage (Fig. 5.8) is a MyShape subclass that represents a JPEG image in a drawing. As we will see in Section 5.6.3, Deitel Drawing enables users to add JPEG images to a drawing using drag and drop. Line 17 declares BufferedImage member variable image for storing the MyImage object’s image.
Chapter 5 Case Study: Java 2D GUI Application with Design Patterns

// set filename for loading image
public void setFileName( String name )
{
    // load image from file
    try {
        File file = new File( name );
        FileInputStream inputStream =
            new FileInputStream( file );
        // decode JPEG image
        JPEGImageDecoder decoder =
            JPEGCodec.createJPEGDecoder( inputStream );
        image = decoder.decodeAsBufferedImage();
        setPoint2( getX1() + image.getWidth(),
                   getY1() + image.getHeight() );
    }
    // handle exception reading image from file
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }
    // set fileName if try is successful
    fileName = name;
}

// get image filename
public String getFileName()
{
    return fileName;
}

// get MyImage XML Element
public Element getXML( Document document )
{
    Element shapeElement = super.getXML( document );
    shapeElement.setAttribute( "type", "MyImage" );
    // create filename Element
    Element temp = document.createElement( "fileName" );
    temp.appendChild( document.createTextNode( getFileName() ) );
    shapeElement.appendChild( temp );
    return shapeElement;
}

Fig. 5.8 MyImage subclass of class MyShape that represents a JPEG image in a drawing (part 2 of 3).
Lines 21–25 implement method `draw` to draw the `MyImage` object. Line 24 invokes method `drawImage` of class `Graphics2D` to draw the `MyImage` object’s `BufferedImage`. Method `contains` (lines 28–34) creates a `Rectangle2D.Float` object of the same dimensions as the `MyImage` object. Line 33 invokes method `contains` of class `Rectangle2D.Float` to determine whether the given `Point2D` object falls within the `MyImage` object’s area.

Method `getImage` (lines 46–49) gets the `BufferedImage` for the `MyImage` object. Method `setFileName` (lines 43–69) takes as a `String` argument the name of the `File` that contains the `MyImage` object’s image. Lines 49–56 open a `FileInputStream` for the `File` and decode the `File` as a JPEG image using method `decodeAsBufferedImage` of class `JPEGImageDecoder`. Line 56 invokes method `setImage` of class `MyImage` to set the `image` property to the newly loaded `BufferedImage`. Method `getFileName` (lines 72–75) returns a `String` that contains the name of the `File` from which the JPEG image was loaded.

Method `getXML` (lines 78–91) creates an XML representation of a `MyImage` object. Line 80 invokes method `getXML` of class `MyShape` to retrieve the default `shape` `Element`. Line 81 adds `Attribute` `type` with the value `MyImage` to `Element` `shape`. Lines 84–87 create a `fileName` `Element` that contains the name of the `File` from which the JPEG image was loaded, and appends this `Element` as a child of `Element` `shape`.

**5.4 Deitel DrawingModel**

The Deitel Drawing application employs the model-view-controller architecture to enhance the application’s modularity and extensibility. Deitel Drawing represents each drawing as a `Collection` of `MyShape` objects stored in a `DrawingModel` (Fig. 5.9). Class
DrawingModel extends Observable (line 13) to allow Observer(s) to register as listeners for changes in the DrawingModel.

```java
// DrawingModel.java
// DrawingModel is the model for a DeitelDrawing painting. It provides methods for adding and removing shapes from a drawing.
package com.deitel.advjhtp1.drawing.model;

// Java core packages
import java.util.*;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class DrawingModel extends Observable {

    // shapes contained in model
    private Collection shapes;

    // no-argument constructor
    public DrawingModel() {
        shapes = new ArrayList();
    }

    // add shape to model
    public void addShape( MyShape shape ) {
        // add new shape to list of shapes
        shapes.add( shape );

        // send model changed notification
        fireModelChanged();
    }

    // remove shape from model
    public void removeShape( MyShape shape ) {
        // remove shape from list
        shapes.remove( shape );

        // send model changed notification
        fireModelChanged();
    }

    // get Collection of shapes in model
    public Collection getShapes() {
        return Collections.unmodifiableCollection( shapes );
    }
}
```

Fig. 5.9 DrawingModel Observable class that represents a drawing containing multiple MyShapes (part 1 of 2).
The **DrawingModel** consists of a **Collection** of **MyShape** objects and methods for adding and removing shapes. Lines 31 and 41 invoke **private** method **fireModelChanged** to notify **Observers** of additions to, and deletions from, the **DrawingModel**. Method **fireModelChanged** (lines 72–79) invokes method **setChanged** of class **Observable** to mark the **DrawingModel** as changed (line 75). Line 78 invokes method **notifyObservers** to send a notification to each registered **Observer** that the **DrawingModel** has changed.

Method **getShapes** (lines 51–54) invokes **static** method **unmodifiableCollection** of class **Collections** to obtain an unmodifiable reference to the **shapes** **Collection**. Returning an unmodifiable **Collection** prevents the caller from changing the model through that **Collection** reference.

Class **DrawingFileReaderWriter** (Fig. 5.10) provides methods **writeFile** and **readFile** for saving and loading drawings. Class **DrawingFileReaderWriter** enables the application to save and load drawings as XML documents. Static method **writeFile** (lines 28–89) takes as arguments a **DrawingModel** and the file name to which the **DrawingModel** should be saved. Lines 34–40 create a new XML DOM object in memory. Lines 43–44 create the **shapes Element**, which is the root of the XML doc-
ument. Lines 47–55 iterate through the DrawingModel’s shapes, and invoke method getXML on each MyShape to obtain its XML Element representation. Line 54 adds each shape Element to the XML document. Lines 58–70 use a Transformer to output the XML document to the given fileName. [Note: If you are not familiar with XML and the Java API for XML Processing, please see Appendices A–D.]

```java
// DrawingFileReaderWriter.java
// DrawingFileReaderWriter defines static methods for reading
// and writing DeitelDrawing files on disk.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.io.*;
import java.util.*;
import java.awt.Color;

// Java extension packages
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// third-party packages
import org.w3c.dom.*;
import org.xml.sax.*;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class DrawingFileReaderWriter {
    // write drawing to file with given fileName
    public static void writeFile( DrawingModel drawingModel,
    String fileName )
    {
        // open file for writing and save drawing data
        try {
            DocumentBuilderFactory builderFactory =
            DocumentBuilderFactory.newInstance();

            DocumentBuilder builder =
            builderFactory.newDocumentBuilder();

            Document document = builder.newDocument();

            // create shapes element to contain all MyShapes
            Element shapesElement =
            document.createElement( "shapes" );
            document.appendChild( shapesElement );
```
Iterator iterator = drawingModel.getShapes().iterator();

// populate shapes element with shape element for each
// MyShape in DrawingModel
while ( iterator.hasNext() ) {
    MyShape shape = ( MyShape ) iterator.next();
    shapesElement.appendChild( shape.getXML( document ) );
}

// use Transformer to write shapes XML document to a file
TransformerFactory transformerFactory =
    TransformerFactory.newInstance();

    Transformer transformer =
    transformerFactory.newTransformer();

    // specify the shapes.dtd Document Type Definition
    transformer.setOutputProperty(
        OutputKeys.DOCTYPE_SYSTEM, "shapes.dtd" );

    transformer.transform( new DOMSource( document ),
        new StreamResult( new FileOutputStream( fileName ) ) );
}

} // end method writeFile

// handle exception building XML Document
catch ( ParserConfigurationException parserException ) {
    parserException.printStackTrace();
}

// handle exception transforming XML Document
catch ( TransformerException transformerException ) {
    transformerException.printStackTrace();
}

// handle exception opening FileOutputStream
catch ( FileNotFoundException fileException ) {
    fileException.printStackTrace();
}
}

} // end method writeFile

// open existing drawing from file
public static Collection readFile( String fileName )
{
    // load shapes from file
    try {
        // Collection of MyShapes read from XML Document

Fig. 5.10  DrawingFileReaderWriter utility class for saving drawings to files and loading drawings from files (part 2 of 8).
```java
Collection shapes = new ArrayList();

DocumentBuilderFactory builderFactory =
    DocumentBuilderFactory.newInstance();

builderFactory.setValidating( true );

DocumentBuilder builder =
    builderFactory.newDocumentBuilder();

Document document = builder.parse(
    new File( fileName ) );

// get all shape elements in XML Document
NodeList list = document.getElementsByTagName( "shape" );

// get MyShape from each shape element in XML Document
for ( int i = 0; i < list.getLength(); i++ ) {
    Element element = ( Element ) list.item( i );
    MyShape shape = getShapeFromElement( element );
    shapes.add( shape );
}

return shapes;

// create MyShape using properties specified in given Element
private static MyShape getShapeFromElement( Element element ) {
    MyShape shape = null;

    // get MyShape type (e.g., MyLine, MyRectangle, etc.)
```
String type = element.getAttribute( "type" );

// create appropriate MyShape subclass instance
if ( type.equals( "MyLine" ) ) {
  shape = new MyLine();
}

else if ( type.equals( "MyRectangle" ) ) {
  shape = new MyRectangle();
}

else if ( type.equals( "MyOval" ) ) {
  shape = new MyOval();
}

else if ( type.equals( "MyText" ) ) {
  shape = new MyText();

  // create MyText reference for setting MyText-specific
  // properties, including fontSize, text, etc.
  MyText textShape = ( MyText ) shape;

  // set text property
  String text =
    getStringValueFromChildElement( element, "text" );
  textShape.setText( text );

  // set fontSize property
  int fontSize =
    getIntValueFromChildElement( element, "fontSize" );
  textShape.setFontSize( fontSize );

  // set fontName property
  String fontName =
    getStringValueFromChildElement( element, "fontName" );
  textShape.setFontName( fontName );

  // set underlined property
  boolean underlined = getBooleanValueFromChildElement( element, "underlined" );
  textShape.setUnderlineSelected( underlined );

  // set bold property
  boolean bold =
    getBooleanValueFromChildElement( element, "bold" );
  textShape.setBoldSelected( bold );
// set italic property
boolean italic =
    getBooleanValueFromChildElement( element, "italic" );

    textShape.setItalicSelected( italic );
}

else if ( type.equals( "MyImage" ) ) {
    shape = new MyImage();

    // create MyImage reference for setting MyImage-specific
    // fileName property
    MyImage imageShape = ( MyImage ) shape;

    String fileName = getStringValueFromChildElement(
        element, "fileName" );

    imageShape.setFileName( fileName );
}

// set properties common to all MyShapes, including x1, y1,
// x2, y2, startColor, endColor, etc.

// set x1 and y1 properties
int x1 = getIntValueFromChildElement( element, "x1" );
int y1 = getIntValueFromChildElement( element, "y1" );

    shape.setPoint1( x1, y1 );

// set x2 and y2 properties
int x2 = getIntValueFromChildElement( element, "x2" );
int y2 = getIntValueFromChildElement( element, "y2" );

    shape.setPoint2( x2, y2 );

// set startX and startY properties
int startX =
    getIntValueFromChildElement( element, "startX" );
int startY =
    getIntValueFromChildElement( element, "startY" );

    shape.setStartPoint( startX, startY );

// set endX and endY properties
int endX = getIntValueFromChildElement( element, "endX" );
int endY = getIntValueFromChildElement( element, "endY" );

    shape.setEndPoint( endX, endY );

// set startColor and endColor properties
Color startColor =
    getColorValueFromChildElement( element, "startColor" );
```java
shape.setStartColor( startColor );

Color endColor =
      getColorValueFromChildElement( element, "endColor" );

shape.setEndColor( endColor );

// set useGradient property
boolean useGradient = getBooleanValueFromChildElement(
      element, "useGradient" );

shape.setUseGradient( useGradient );

// set strokeSize property
float strokeSize = getFloatValueFromChildElement( 
      element, "strokeSize" );

shape.setStrokeSize( strokeSize );

// set filled property
boolean fill =
      getBooleanValueFromChildElement( element, "fill" );

shape.setFilled( fill );

return shape;

} // end method getShapeFromElement

// get int value from child element with given name
private static int getIntValueFromChildElement( Element parent, 
      String childElementName )
{
      // get NodeList for Elements of given childElementName
      NodeList childNodes = parent.getElementsByTagName( 
          childElementName );

      // get Text Node from zeroth child Element
      Node childTextNode = childNodes.item( 0 ).getFirstChild();

      // parse int value from Text Node
      return Integer.parseInt( childTextNode.getNodeValue() );

} // end method getIntValueFromChildElement

// get float value from child element with given name
private static float getFloatValueFromChildElement( 
      Element parent, String childElementName )
{
      // get NodeList for Elements of given childElementName
      NodeList childNodes = parent.getElementsByTagName( 
          childElementName );

      // get Text Node from zeroth child Element
      Node childTextNode = childNodes.item( 0 ).getFirstChild();

      // parse float value from Text Node
      return Float.parseFloat( childTextNode.getNodeValue() );

} // end method getFloatValueFromChildElement
```

---

**Fig. 5.10** **DrawingFileReaderWriter** utility class for saving drawings to files and loading drawings from files (part 6 of 8).
childElementName );

// get Text Node from zeroth child Element
Node childTextNode = childNodes.item( 0 ).getFirstChild();

// parse float value from Text Node
return Float.parseFloat( childTextNode.getNodeValue() );

} // end method getFloatValueFromChildElement

// get boolean value from child element with given name
private static boolean getBooleanValueFromChildElement(
    Element parent, String childElementName )
{
    // get NodeList for Elements of given childElementName
    NodeList childNodes = parent.getElementsByTagName(
        childElementName );

    Node childTextNode = childNodes.item( 0 ).getFirstChild();

    // parse boolean value from Text Node
    return Boolean.valueOf( childTextNode.getNodeValue() ).booleanValue();

} // end method getBooleanValueFromChildElement

// get String value from child element with given name
private static String getStringValueFromChildElement(
    Element parent, String childElementName )
{
    // get NodeList for Elements of given childElementName
    NodeList childNodes = parent.getElementsByTagName(
        childElementName );

    // get Text Node from zeroth child Element
    Node childTextNode = childNodes.item( 0 ).getFirstChild();

    // return String value of Text Node
    return childTextNode.getNodeValue();

} // end method getStringValueFromChildElement

// get Color value from child element with given name
private static Color getColorValueFromChildElement(
    Element parent, String childElementName )
{
    // get NodeList for Elements of given childElementName
    NodeList childNodes = parent.getElementsByTagName(
        childElementName );

    // get zeroth child Element
    Element childElement = ( Element ) childNodes.item( 0 );

Fig. 5.10 DrawingFileReaderWriter utility class for saving drawings to files and loading drawings from files (part 7 of 8).
Method `readFile` (lines 92–142) loads a drawing from an XML document. Lines 100–109 create a `DocumentBuilder` and parse the XML document with the given `fileName`. Line 112 invokes method `getElementsByTagName` of interface `Document` to retrieve all `shape` `Element`s in the document. Lines 115–119 process each `shape` `Element` by invoking method `getShapeFromElement` (line 117), which returns a `MyShape` object for each `Element`. Line 120 adds each `MyShape` to the `shapesCollection`.

Method `getShapeFromElement` (lines 145–282) builds an appropriate `MyShape` subclass instance for the given `shapeElement`. Line 150 retrieves the value of the `type` `Attribute` to determine the appropriate `MyShape` subclass to instantiate. Lines 170–206 obtain values specific to `MyText` objects. Lines 216–219 obtain values specific to `MyImage` objects. Lines 226–278 obtain values that apply to all `MyShapes`.

Method `getIntValueFromChildElement` (lines 285–298) is a utility method for obtaining an `int` value from a particular child `Element`. Lines 289–290 obtain a `NodeList` of `Elements` with the given `childElementName`. Line 293 obtains the `Text Node` child of the `Element` and line 298 parses the `Text Node` to produce an `int` value. Methods `getFloatValueFromChildElement` (lines 301–314), `getBooleanValueFromChildElement` (lines 317–330), `getStringValueFromChildElement` (lines 333–346) and `getColorValueFromChildElement` perform similar processing to retrieve values of other data types.

Figure 5.11 shows a sample XML document produced by `DrawingFileReaderWriter`. Note that the `MyText shape` element (lines 49–69) has child elements `text`, `fontSize`, `fontName`, `underline`, `bold` and `italic`, whereas the other shape elements have only the basic `MyShape-related elements.`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE shapes SYSTEM "shapes.dtd">
```

**Fig. 5.10** `DrawingFileReaderWriter` utility class for saving drawings to files and loading drawings from files (part 8 of 8).

**Fig. 5.11** Sample XML document generated by `DrawingFileReaderWriter` (part 1 of 3).
<shapes>
  <shape type="MyLine">
    <x1>122</x1>
    <y1>36</y1>
    <x2>43</x2>
    <y2>120</y2>
    <startX>43</startX>
    <startY>120</startY>
    <endX>122</endX>
    <endY>36</endY>
    <useGradient>false</useGradient>
    <startColor red="0" green="0" blue="0"/>
    <endColor red="255" green="255" blue="255"/>
    <strokeSize>1.0</strokeSize>
    <fill>false</fill>
  </shape>
  <shape type="MyRectangle">
    <x1>62</x1>
    <y1>71</y1>
    <x2>124</x2>
    <y2>132</y2>
    <startX>62</startX>
    <startY>71</startY>
    <endX>124</endX>
    <endY>132</endY>
    <useGradient>false</useGradient>
    <startColor red="0" green="0" blue="0"/>
    <endColor red="255" green="255" blue="255"/>
    <strokeSize>1.0</strokeSize>
    <fill>false</fill>
  </shape>
  <shape type="MyOval">
    <x1>18</x1>
    <y1>11</y1>
    <x2>107</x2>
    <y2>123</y2>
    <startX>18</startX>
    <startY>11</startY>
    <endX>107</endX>
    <endY>123</endY>
    <useGradient>false</useGradient>
    <startColor red="0" green="0" blue="0"/>
    <endColor red="255" green="255" blue="255"/>
    <strokeSize>1.0</strokeSize>
    <fill>false</fill>
  </shape>
  <shape type="MyText">
    <x1>38</x1>
    <y1>167</y1>
    <x2>0</x2>
    <y2>0</y2>
    <startX>0</startX>
  </shape>
</shapes>

Fig. 5.11  Sample XML document generated by DrawingFileReaderWriter (part 2 of 3).
5.5 Deitel Drawing Views

The Deitel Drawing application provides two views of user drawings. Class **DrawingView** (Fig. 5.12) is the primary view and extends **JPanel** to provide a surface onto which the user can draw **MyShapes**. Class **DrawingView** also implements interface **Observer** (line 20), so it can listen for **DrawingModel** changes.

```java
// DrawingView.java
// DrawingView is a view of a DrawingModel that draws shapes using the Java2D API.
package com.deitel.advjhtp1.drawing.view;

// Java core packages
import java.awt.*;
```

**Fig. 5.11** Sample XML document generated by **DrawingFileReaderWriter** (part 3 of 3).

**Fig. 5.12** **DrawingView** class for displaying **MyShapes** in a **DrawingModel** (part 1 of 4).
import java.awt.geom.*;  
import java.awt.event.*;  
import java.util.*;  
import java.util.List;  

// Java extension packages  
import javax.swing.*;  

// Deitel packages  
import com.deitel.advjhtp1.drawing.model.*;  
import com.deitel.advjhtp1.drawing.model.shapes.*;  

public class DrawingView extends JPanel implements Observer {
    private DrawingModel drawingModel;  

    public DrawingView( DrawingModel model ) {
        // set DrawingModel  
        drawingModel = model;  

        // set background color  
        setBackground( Color.white );  

        // enable double buffering to reduce screen flicker  
        setDoubleBuffered( true );  
    }

    public void setModel( DrawingModel model ) {
        if ( drawingModel != null )  
            drawingModel.deleteObserver( this );

        drawingModel = model;  

        if ( model != null ) {  
            model.addObserver( this );  
            repaint();  
        }
    }

    public DrawingModel getModel() {
        return drawingModel;  
    }

    // repaint view when update received from model

Fig. 5.12 DrawingView class for displaying MyShapes in a DrawingModel (part 2 of 4).
public void update( Observable observable, Object object )
{
    repaint();
}

// overridden paintComponent method for drawing shapes
public void paintComponent( Graphics g )
{
    // call superclass paintComponent
    super.paintComponent( g );

    // create Graphics2D object for given Graphics object
    Graphics2D g2D = ( Graphics2D ) g;

    // enable anti-aliasing to smooth jagged lines
    g2D.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                          RenderingHints.VALUE_ANTIALIAS_ON );

    // enable high-quality rendering in Graphics2D object
    g2D.setRenderingHint( RenderingHints.KEY_RENDERING,
                          RenderingHints.VALUE_RENDER_QUALITY );

    // draw all shapes in model
    drawShapes( g2D );
}

// draw shapes in model
public void drawShapes( Graphics2D g2D )
{
    // get Iterator for shapes in model
    Iterator iterator = drawingModel.getShapes().iterator();

    // draw each MyShape in DrawingModel
    while ( iterator.hasNext() ) {
        MyShape shape = ( MyShape ) iterator.next();
        shape.draw( g2D );
    }
}

// get preferred size for this component
public Dimension getPreferredSize()
{
    return new Dimension( 320, 240 );
}

// insist on preferred size for this component
public Dimension getMinimumSize()
{
    return getPreferredSize();
}

// insist on preferred size for this component

Fig. 5.12 DrawingView class for displaying MyShapes in a DrawingModel (part 3 of 4).
Method `setModel` (lines 39–51) first removes the `DrawingView` as an Observer of the existing `DrawingModel` (line 42), then registers the `DrawingView` as an Observer for the new `DrawingModel` (line 48). The `ObservableDrawingModel` invokes method `update` of class `DrawingView` (lines 60–63) each time the `DrawingModel` changes. Method `update` invokes method `repaint` of class `JPanel` (line 62) each time the `DrawingView` receives an update from the `DrawingModel`. Methods `addNotify` (lines 119–123) and `removeNotify` (lines 127–131) add and delete the `DrawingView` as an Observer of the `DrawingModel` when the `DrawingView` obtains and discards its screen resources, respectively.

Method `paintComponent` (lines 66–84) configures the `Graphics2D` context for high-quality, anti-aliased drawing (lines 75–80) and invokes method `drawShapes` (line 83) to draw the `DrawingModel`’s shapes. Method `drawShapes` (lines 87–101) gets an `Iterator` for the `Collection` of `MyShapes` obtained from the `DrawingModel` (line 90). Lines 93–96 draw each `MyShape` on the given `Graphics2D` context.

Class `ZoomDrawingView` (Fig. 5.13) extends class `DrawingView` to provide a scaled view of a `DrawingModel`. Line 21 declares an `AffineTransform` reference that `ZoomDrawingView` uses to scale its rendering of the `DrawingModel`. The primary `ZoomDrawingView` constructor (lines 38–67) takes as arguments a `DrawingModel` and the factors by which the `AffineTransform` should scale points along the x- and y-axes.

Lines 48–65 add a `ComponentListener` anonymous inner class for the `ZoomDrawingView`. This `ComponentListener` adjusts the scale factors when the `ZoomDrawingView` component changes size. This allows the user to resize a window that contains a `ZoomDrawingView` to change its scale. For example, if the user resizes the window to 640x480—twice the size of a default `DrawingView`—the `AffineTransform` magnifies the drawing view by a scale factor of 2. If the user resizes the window to
160x120—half the size of a default.DrawingView—the AffineTransform shrinks the drawing view by a scale factor of 0.5. The x- and y-axes also scale independently. The user can stretch the window horizontally to produce a short, wide drawing view or vertically to produce a tall, narrow drawing view.

```java
// ZoomDrawingView.java
// ZoomDrawingView is a subclass of DrawingView that scales
// the view of the drawing using the given scale factor.
package com.deitel.advjhtp1.drawing.view;

// Java core packages
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.*;

public class ZoomDrawingView extends DrawingView {

    // factor for scaling view
    private double scaleFactorX;
    private double scaleFactorY;

    // transform for scaling view
    private AffineTransform scaleTransform;

    // construct ZoomDrawingView with given model and default
    // scale factor
    public ZoomDrawingView( DrawingModel model )
    {
        this( model, 1.0 );
    }

    // construct ZoomDrawingView with given model and scale factor
    public ZoomDrawingView( DrawingModel model, double scale )
    {
        this( model, scale, scale );
    }

    // construct ZoomDrawingView with given model and separate
    // x and y scale factors
    public ZoomDrawingView( DrawingModel model, double scaleX,
                            double scaleY )
    {
        // call DrawingView constructor
        super( model );

        // set scale factor for this view
        setScaleFactors( scaleX, scaleY );
    }
}
```

*Fig. 5.13*  ZoomDrawingView subclass of DrawingView for displaying scaled MyShapes (part 1 of 3).
47 // listen for component resize events to adjust scale
48 addComponentListener(
49     new ComponentAdapter() {
50         // when view is resized, update scale factors
51         public void componentResized( ComponentEvent event )
52             {
53                 double width = ( double ) getSize().width;
54                 double height = ( double ) getSize().height;
55                 // calculate new scale factors
56                 double factorX = width / 320.0;
57                 double factorY = height / 240.0;
58                setScaleFactors( factorX, factorY );
59             }
60         } // end ZoomDrawingView constructor
61     } // end ZoomDrawingView subclass of DrawingView for displaying scaled MyShapes (part 2 of 3).
62 }
63
64 // draw shapes using scaled Graphics2D object
65 public void drawShapes( Graphics2D g2D )
66 {
67     // set Graphics2D object transform
68     g2D.setTransform( scaleTransform );
69     // draw shapes on scaled Graphics2D object
70     super.drawShapes( g2D );
71 }
72
73 // set scale factors for view
74 public void setScaleFactors( double scaleX, double scaleY )
75 {
76     // set scale factors
77     scaleFactorX = scaleX;
78     scaleFactorY = scaleY;
79     // create AffineTransform with given scale factors
80     scaleTransform = AffineTransform.getScaleInstance(
81             scaleFactorX, scaleFactorY );
82 }
83
84 // get preferred size for this component
85 public Dimension getPreferredSize()
86 {
87     // default size is 320 x 240; scale using scaleFactors
88     return new Dimension( ( int ) ( 320 * scaleFactorX ),
89             ( int ) ( 240 * scaleFactorY ) );
90 }
Method `drawShapes` (lines 70–77) overrides method `drawShapes` from class `DrawingView`. Line 73 invokes method `setTransform` of class `Graphics2D` to cause the `Graphics2D` object to use the provided `AffineTransform` to scale the drawing. Method `setScaleFactors` (lines 80–89) takes as `double` arguments the scale factors to use for the x- and y-axes. Lines 87–88 create the `AffineTransform` that method `drawShapes` uses to scale the drawing. Static method `getScaleInstance` of class `AffineTransform` returns an `AffineTransform` object that scales drawings based on the provided x- and y-axis scale factors. For example, scale factors of 0.5 and 0.5 would produce a view that is one quarter the original size.

### 5.6 Deitel Drawing Controller Logic

The model-view-controller architecture separates logic for processing user input into objects that are separate from the views and the model. The Deitel Drawing application uses two types of controllers to handle user input—`MyShapeController`s and a `DragAndDropController`.

#### 5.6.1 MyShapeControllers for Processing User Input

The primary user-input device for creating drawings is the mouse. A user can create and manipulate new shapes in a drawing by pressing the mouse button, dragging the mouse then releasing the mouse button. For each type of `MyShape`, however, there are different requirements for handling mouse events. For example, drawing a `MyText` shape requires the application to obtain from the user the text to be drawn and that text’s properties, such as its font size. Class `MyShapeController` (Fig. 5.14) is an abstract base class that defines the basic functionality required by all `MyShapeControllers`. Subclasses of `MyShapeController` provide the implementation details for adding instances of each particular `MyShape` subclass to a drawing.
// MyShapeController.java
// MyShapeController is an abstract base class that represents
// a controller for painting shapes.
package com.deitel.advjhtp1.drawing.controller;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;

public abstract class MyShapeController {

    private DrawingModel drawingModel;

    // primary and secondary Colors for drawing and gradients
    private Color primaryColor = Color.black;
    private Color secondaryColor = Color.white;

    // Class object for creating new MyShape-subclass instances
    private Class shapeClass;

    // common MyShape properties
    private boolean fillShape = false;
    private boolean useGradient = false;
    private float strokeSize = 1.0f;

    // indicates whether the user has specified drag mode; if true, MyShapeController should ignore mouse events
    private boolean dragMode = false;

    private MouseListener mouseListener;
    private MouseMotionListener mouseMotionListener;

    // MyShapeController constructor
    public MyShapeController( DrawingModel model, Class myShapeClass ) {
        // set DrawingModel to control
        drawingModel = model;

        // set MyShape subclass
        shapeClass = myShapeClass;

        // listen for mouse events
        mouseListener = new MouseAdapter() {
            // when mouse button pressed, create new shape
            public void mousePressed( MouseEvent event ) {
                // ...
            }
        };
    }

}
// if not in dragMode, start new shape at
// given coordinates
if ( !dragMode )
    startShape( event.getX(), event.getY() );

// when mouse button released, set shape's final
// coordinates
public void mouseReleased( MouseEvent event )
{
    // if not in dragMode, finish drawing current shape
    if ( !dragMode )
        endShape( event.getX(), event.getY() );
}

// listen for mouse motion events
mouseMotionListener = new MouseMotionAdapter() {
    // when mouse is dragged, set coordinates for current
    // shape's Point2
    public void mouseDragged( MouseEvent event )
    {
        // if not in dragMode, modify current shape
        if ( !dragMode )
            modifyShape( event.getX(), event.getY() );
    }
};

// set primary color (start color for gradient)
public void setPrimaryColor( Color color )
{
    primaryColor = color;
}

// get primary color
public Color getPrimaryColor()
{
    return primaryColor;
}

// set secondary color (end color for gradients)
public void setSecondaryColor( Color color )
{
    secondaryColor = color;
}

// get secondary color
public Color getSecondaryColor()
{

Fig. 5.14  MyShapeController abstract base class for controllers that handle mouse input (part 2 of 5).
105      return secondaryColor;
106    }
107
108    // fill shape
109    public void setShapeFilled( boolean fill )
110    {
111      fillShape = fill;
112    }
113
114    // get shape filled
115    public boolean getShapeFilled()
116    {
117      return fillShape;
118    }
119
120    // use gradient when painting shape
121    public void setUseGradient( boolean gradient )
122    {
123      useGradient = gradient;
124    }
125
126    // get use gradient
127    public boolean getUseGradient()
128    {
129      return useGradient;
130    }
131
132    // set dragMode
133    public void setDragMode( boolean drag )
134    {
135      dragMode = drag;
136    }
137
138    // set stroke size for lines
139    public void setStrokeSize( float stroke )
140    {
141      strokeSize = stroke;
142    }
143
144    // get stroke size
145    public float getStrokeSize()
146    {
147      return strokeSize;
148    }
149
150    // create new instance of current MyShape subclass
151   protected MyShape createNewShape()
152    {
153      // create new instance of current MyShape subclass
154      try {
155        MyShape shape = ( MyShape ) shapeClass.newInstance();
156      } catch (Exception e) {
157        // handle exception
158      }
159    }
// set MyShape properties
shape.setFilled( fillShape );
shape.setUseGradient( useGradient );
shape.setStrokeSize( getStrokeSize() );
shape.setStartColor( getPrimaryColor() );
shape.setEndColor( getSecondaryColor() );

// return reference to newly created shape
return shape;

// handle exception instantiating shape
try {
    instanceException.printStackTrace();
    return null;
} catch (InstantiationException instanceException) {

}

// handle access exception instantiating shape
try {
    accessException.printStackTrace();
    return null;
} catch (IllegalAccessException accessException) {

}

} // end method createNewShape

// get MyShapeController's MouseListener
public MouseListener getMouseListener() {
    return mouseListener;
}

// get MyShapeController's MouseMotionListener
public MouseMotionListener getMouseMotionListener() {
    return mouseMotionListener;
}

// add given shape to DrawingModel
protected void addShapeToModel( MyShape shape ) {
    drawingModel.addShape( shape );
}

// remove given shape from DrawingModel
protected void removeShapeFromModel( MyShape shape ) {
    drawingModel.removeShape( shape );
}

// start new shape
public abstract void startShape( int x, int y );
Each \texttt{MyShapeController} is responsible for responding to mouse events to allow users to add shapes to drawings. Lines 48–67 create a \texttt{MouseListener} that listens for \texttt{mousePressed} and \texttt{mouseReleased} events. When the user presses the mouse button, line 56 starts drawing a new shape at the location where the mouse press occurred by invoking method \texttt{startShape}. When the user releases the mouse button, line 65 invokes method \texttt{endShape} to complete the currently drawn shape. As the user drags the mouse, the \texttt{MouseMotionListener} on lines 70–80 invokes method \texttt{modifyShape} to modify the shape currently being drawn.

Note that class \texttt{MyShapeController} uses instances of \texttt{MouseAdapter} and \texttt{MouseMotionAdapter} to respond to \texttt{MouseEvent}s. Objects of the classes \texttt{MouseAdapter} and \texttt{MouseMotionAdapter} act as adapters between objects that generate \texttt{MouseEvent}s and those objects that handle these events. In this case study, \texttt{MyShapeController}’s \texttt{MouseAdapter} (lines 46–65) and \texttt{MouseMotionAdapter} (lines 68–79) adapts a \texttt{MyShapeController} to a \texttt{MouseListener} and \texttt{MouseMotionListener}, respectively. These adapter classes are examples of the Adapter design pattern, which provides an object with a new interface that adapts to another object’s interface, allowing both objects to collaborate with one another. The adapter in this pattern is similar to an adapter for a plug on an electrical device—electrical sockets in Europe are different from those in the United States, so an adapter is needed to plug an American device into a European electrical socket and vice versa.

Methods \texttt{startShape}, \texttt{endShape} and \texttt{modifyShape} are abstract methods that each \texttt{MyShapeController} subclass must implement. This enables the developer to provide custom controllers for different shape types. The developer simply overrides these methods to perform the necessary input processing logic.

Method \texttt{createNewShape} (lines 151–180) uses Java’s reflection mechanism to create new instances of \texttt{MyShape} subclasses as the user adds new shapes to a drawing. Reflection enables Java programs to determine information about classes and objects at runtime. In this example, we use reflection to enable our application to create instances of arbitrary \texttt{MyShape} subclasses dynamically. Each \texttt{MyShapeController} maintains a \texttt{Class} reference to the \texttt{Class} object for the \texttt{MyShape} subclass that the \texttt{MyShapeController} controls. For example, when the application creates a \texttt{MyShapeController} for drawing \texttt{MyLines}, the \texttt{MyShapeController} stores a reference to the \texttt{Class} object for class \texttt{MyLine}. Line 155 invokes method \texttt{newInstance} of class \texttt{Class} to create a new instance of the specified \texttt{MyShape} subclass. Lines 158–162 initialize this new instance with the currently selected fill, gradient, stroke size and color properties.

Class \texttt{BoundedShapeController} (Fig. 5.15) provides a basic implementation of abstract base class \texttt{MyShapeController} for drawing rectangle-bounded shapes (in this
application, **MyRectangles** and **MyOval**s). Method **startShape** (lines 22–37) creates a new instance of the appropriate **MyShape** subclass (line 25), sets the **MyShape**’s position on the drawing (lines 30–32) and adds the **MyShape** to the **DrawingModel** (line 35). The **MouseListener** in class **MyShapeController** invokes method **startShape** when the user presses the mouse button to begin drawing a shape.

```java
// BoundedShapeController.java
// BoundedShapeController is a MyShapeController subclass for
// rectangle-bounded shapes, such as MyOvals and MyRectangles.
package com.deitel.advjhtp1.drawing.controller;

import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class BoundedShapeController extends MyShapeController {

    private MyShape currentShape;

    // BoundedShapeController constructor
    public BoundedShapeController(
        DrawingModel model, Class shapeClass )
    {
        super( model, shapeClass );
    }

    // start drawing shape
    public void startShape( int x, int y )
    {
        // get new shape
        currentShape = createNewShape();

        if ( currentShape != null ) {
            // set location of shape in drawing
            currentShape.setPoint1( x, y );
            currentShape.setPoint2( x, y );
            currentShape.setStartPoint( x, y );

            // add newly created shape to DrawingModel
            addShapeToModel( currentShape );
        }
    }
    
    // modify shape currently being drawn
    public void modifyShape( int x, int y )
    {
        // remove shape from DrawingModel
        removeShapeFromModel( currentShape );
        currentShape.setEndPoint( x, y );
    }
}
```

**Fig. 5.15** **BoundedShapeController** **MyShapeController** subclass for controlling **MyLines**, **MyOvals** and **MyRectangles** (part 1 of 2).
When the user drags the mouse, the `MouseMotionListener` inherited from class `MyShapeController` invokes method `modifyShape` (lines 40–59) and passes the x- and y-coordinates of the `MouseEvent`. Method `modifyShape` removes `currentShape` from the `DrawingModel` (line 43), updates the `currentShape`’s various point properties with new coordinates (lines 46–55) and adds `currentShape` to the `DrawingModel`.

When the user releases the mouse button, the mouse handler invokes method `endShape` to complete the addition of the `MyShape` to the drawing. Method `endShape` invokes method `modifyShape` (line 52) to set the final values for `currentShape`’s coordinates.

Class `MyLineController` (Fig. 5.16) is a `MyShapeController` subclass for drawing `MyLine` objects. Method `startShape` (lines 20–36) is similar to method `startShape` in class `BoundedShapeController`. Method `modifyShape` (lines 39–56) removes the `MyLine` from the `DrawingModel` (line 42) and sets the `MyLine`’s `endPoint` to the current x, y coordinate. Lines 49–52 update the `MyLine`’s `Point1` and `Point2` coordinates.
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class MyLineController extends MyShapeController {

    private MyShape currentShape;

    // MyLineController constructor
    public MyLineController( DrawingModel model, Class shapeClass )
    {
        super( model, shapeClass );
    }

    // start drawing new shape
    public void startShape( int x, int y )
    {
        // create new shape
        currentShape = createNewShape();

        if ( currentShape != null ) {
            // set location of shape in drawing
            currentShape.setPoint1( x, y );
            currentShape.setPoint2( x, y );
            currentShape.setStartPoint( x, y );

            // add newly created shape to DrawingModel
            addShapeToModel( currentShape );
        }
    } // end method startShape

    // modify shape currently being drawn
    public void modifyShape( int x, int y )
    {
        // remove shape from DrawingModel
        removeShapeFromModel( currentShape );
        currentShape.setEndPoint( x, y );

        int startX = currentShape.getStartX();
        int startY = currentShape.getStartY();

        // set current ( x, y ) to Point1
        currentShape.setPoint1( x, y );

        // set Point2 to StartPoint
        currentShape.setPoint2( startX, startY );

        // add shape back into model
        addShapeToModel( currentShape );
    }

    // finish drawing shape

Fig. 5.16  MyLineController MyShapeController subclass for drawing MyLines (part 2 of 3).
Instances of class MyText are drawn quite differently from instances of classes MyLine, MyOval and MyRectangle and therefore require a custom implementation of class MyShapeController. Class MyTextController (Fig. 5.17) presents a dialog box that prompts the user for the text to be drawn as well as MyText properties (e.g., bold, italic, font, etc.).

Fig. 5.16  MyLineController MyShapeController subclass for drawing MyLines (part 3 of 3).

Fig. 5.17  MyTextController MyShapeController subclass for adding MyText instances to a drawing (part 1 of 4).
// create TextInputPanel to get text and properties
TextInputPanel inputPanel = new TextInputPanel();

// display TextInputPanel in JOptionPane
String text = JOptionPane.showInputDialog( null, inputPanel );

// ensure provided text is not null or empty
if ( text == null || text.equals( "" ) )
    return;

// set MyText properties (bold, italic, etc.)
currentText.setBoldSelected( inputPanel.boldSelected() );
currentText.setItalicSelected( inputPanel.italicSelected() );
currentText.setUnderlineSelected( inputPanel.underlineSelected() );
currentText.setFontName( inputPanel.getSelectedFontName() );
currentText.setFontSize( inputPanel.getSelectedFontSize() );
currentText.setColor( getPrimaryColor() );

// set MyText's text
currentText.setText( text );

// add MyText object to model
addShapeToModel( currentText );
}

// modify shape currently being drawn
public void modifyShape( int x, int y ) {} 

// finish drawing shape
public void endShape( int x, int y ) {}

// JPanel with components for inputting MyText properties
private static class TextInputPanel extends JPanel {
    private JCheckBox boldCheckBox;
    private JCheckBox italicCheckBox;
    private JCheckBox underlineCheckBox;
    private JComboBox fontComboBox;
    private JComboBox fontSizeComboBox;
    
    // TextInputPanel constructor

Fig. 5.17 MyTextController MyShapeController subclass for adding MyText instances to a drawing (part 2 of 4).
```java
public TextInputPanel()
{
    boldCheckBox = new JCheckBox( "Bold" );
    italicCheckBox = new JCheckBox( "Italic" );
    underlineCheckBox = new JCheckBox( "Underline" );

    // create JComboBox for selecting Font
    fontComboBox = new JComboBox();
    fontComboBox.addItem( "SansSerif" );
    fontComboBox.addItem( "Serif" );

    // create JComboBox for selecting Font size
    fontSizeComboBox = new JComboBox();
    fontSizeComboBox.addItem( "10" );
    fontSizeComboBox.addItem( "12" );
    fontSizeComboBox.addItem( "14" );
    fontSizeComboBox.addItem( "18" );
    fontSizeComboBox.addItem( "22" );
    fontSizeComboBox.addItem( "36" );
    fontSizeComboBox.addItem( "48" );
    fontSizeComboBox.addItem( "72" );

    setLayout( new FlowLayout() );

    add( boldCheckBox );
    add( italicCheckBox );
    add( underlineCheckBox );
    add( fontComboBox );
    add( fontSizeComboBox );
}

// get bold property
public boolean boldSelected()
{
    return boldCheckBox.isSelected();
}

// get italic property
public boolean italicSelected()
{
    return italicCheckBox.isSelected();
}

// get underline property
public boolean underlineSelected()
{
    return underlineCheckBox.isSelected();
}

// get font name property
public String getSelectedFontName()
{
    Fig. 5.17 MyTextController MyShapeController subclass for adding MyText instances to a drawing (part 3 of 4).
}
Method \texttt{startShape} (lines 27–69) creates a new \texttt{MyText} object (line 30) and sets its coordinates to the given $x$, $y$ coordinate. Line 36 creates a new \texttt{TextInputPanel} and lines 39–40 display the \texttt{TextInputPanel} in a \texttt{JOptionPane}. After the user enters the text and its properties, lines 47–68 set the \texttt{MyText} object’s properties and add the \texttt{MyText} object to the \texttt{DrawingModel}. \texttt{MyTextController} uses static class \texttt{TextInputPanel} (lines 78–148) to present a GUI for setting \texttt{MyText} object properties. Class \texttt{TextInputPanel} includes \texttt{JCheckBox}es for selecting bold, italic and underline, and \texttt{JComboBox}es for selecting the font and font size.

Class \texttt{MyShapeController} uses the Template Method design pattern to ensure that all \texttt{MyShapeControllers} follow the same three-step algorithm for creating shapes—the user clicks on the drawing area to specify a shape’s position, drags the mouse cursor across the area to specify its size, then releases the mouse button to create the shape. These steps correspond to the abstract methods \texttt{startShape}, \texttt{modifyShape} and \texttt{endShape}, respectively. Each \texttt{MyShapeController} subclass uses this algorithm but implements each step differently from the other implementations. For example, method \texttt{startShape} of class \texttt{MyTextController} presents a dialog box that obtains font information and text from the user. However, neither class \texttt{MyLineController} nor class \texttt{BoundedShapeController} needs to set fonts, so their implementations of method \texttt{startShape} differ from that of class \texttt{MyTextController}. Because the Template Method design pattern encapsulates a step-by-step algorithm that several objects can use, this pattern becomes beneficial when we add new \texttt{MyShapeController} subclasses (e.g., \texttt{RandomShapeController} in Exercise 5.8) to our system—we need implement only those methods that comprise the algorithm.

\subsection*{5.6.2 \texttt{MyShapeControllers} and Factory Method Design Pattern}

The model-view-controller architecture makes the Deitel Drawing application easily extensible through the addition of new \texttt{MyShape} subclasses and new views. Creating a new \texttt{MyShape} subclass also could require a new \texttt{MyShapeController} subclass (as class \texttt{MyText} does). To eliminate the need to change existing code when adding a new \texttt{MyShapeController} to the application, Deitel Drawing uses a combination of two popular design patterns—the Factory Method design pattern and the Singleton design pattern.
The Deitel Drawing application uses the *Factory Method design pattern* to enable the user to select an appropriate `MyShapeController` at runtime. As its name implies, a Factory method creates objects. Factory methods can create objects based on criteria that are known only at runtime. These criteria could be in the form of user input, system properties, etc. In the Deitel Drawing application, the criterion is the user-selected `MyShape` type. The particular `MyShape` subclass the user selects is known only at run time, so at compile time we cannot determine what type of `MyShapeController` to use for controlling user input. We use a Factory Method in class `MyShapeControllerFactory` (Fig. 5.18) to construct the appropriate `MyShapeController` for the user-selected `MyShape`.

Method `newMyShapeController` (lines 75–106) is a Factory Method that takes as a `DrawingModel` argument the model to be controlled and as a `String` argument the name of the `MyShape` subclass for which to create a `MyShapeController` instance. Lines 83–85 invoke static method `forName` of class `Class` to get the `Class` object for the given `MyShape` subclass. If the given `MyShape` subclass is `MyLine`, line 89 returns a `MyLineController`. If the given `MyShape` subclass is `MyText`, line 92 returns a new instance of class `MyTextController`. Otherwise, lines 95–96 return a new instance of class `BoundedShapeController`.

```java
// MyShapeControllerFactory.java
// MyShapeControllerFactory uses the Factory Method design
// pattern to create an appropriate instance of MyShapeController
// for the given MyShape subclass.
package com.deitel.advjhtp1.drawing.controller;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class MyShapeControllerFactory {
    private static final String FACTORY_PROPERTY_KEY =
        "MyShapeControllerFactory";

    private static final String[] supportedShapes =
        { "MyLine", "MyRectangle", "MyOval", "MyText" };

    private static MyShapeControllerFactory factory;

    // MyShapeControllerFactory constructor
    protected MyShapeControllerFactory() {
    }

    // return Singleton instance of MyShapeControllerFactory
    public static final MyShapeControllerFactory getInstance()
    {  // if factory is null, create new MyShapeControllerFactory
        if ( factory == null ) {
```
// get System property that contains the factory
// class name
String factoryClassName =
    System.getProperty( FACTORY_PROPERTY_KEY );

// if the System property is not set, create a new
// instance of the default MyShapeControllerFactory
if ( factoryClassName == null )
    factory = new MyShapeControllerFactory();

// create a new MyShapeControllerFactory using the
// class name provided in the System property
else {

    // create MyShapeControllerFactory subclass instance
    try {
        factory = ( MyShapeControllerFactory )
            Class.forName( factoryClassName ).newInstance();
    }

    // handle exception loading instantiating
catch ( ClassNotFoundException classException ) {
        classException.printStackTrace();
    }

    // handle exception instantiating factory
catch ( InstantiationException exception ) {
        exception.printStackTrace();
    }

    // handle exception if no access to specified Class
catch ( IllegalAccessException accessException ) {
        accessException.printStackTrace();
    }

    } // end if

    return factory;

} // end method getInstance

// create new MyShapeController subclass instance for given
// suitable for controlling given MyShape subclass type
public MyShapeController newMyShapeController( 
    DrawingModel model, String shapeClassName )
{
    // create Class instance for given class name and
    // construct appropriate MyShapeController
    try {
        // ...
Class `MyShapeControllerFactory` also uses the Singleton design pattern to control how other objects obtain instances of `MyShapeControllerFactory`. Specifically, the Singleton design pattern ensures that only one instance of a particular object can exist in a particular application. Class `MyShapeControllerFactory` declares a protected, no-argument constructor to prevent other objects from instantiating `MyShapeControllerFactory` objects directly. Other objects that require an instance of class `MyShapeControllerFactory` can invoke method `getInstance` (lines 26–71) to obtain the Singleton instance. If a `MyShapeControllerFactory` has not been created yet, lines 29–65 create a `MyShapeControllerFactory`. Line 69 returns the Singleton `MyShapeControllerFactory` instance to the caller.

In this example, our implementation provides the benefit of allowing the `MyShapeControllerFactory` to determine at runtime the particular subclass of `MyShapeControllerFactory` to instantiate. Lines 33–34 read a system property whose value specifies from which particular `MyShapeControllerFactory` subclass method `getInstance` should instantiate a new `MyShapeControllerFactory`. By speci-
fying a value for this system property at the command line, a user can “install” a new `MyShapeControllerFactory` subclass without requiring changes to existing application code. For example, Exercise 5.8 asks you to create a `RandomMyShapeController` that draws random shapes in the drawing. To add this new `MyShapeControllerFactory` to the application, you also must create a `MyShapeControllerFactory` subclass (e.g., `RandomMyShapeControllerFactory`) that creates `RandomMyShapeControllers`. From the command line, the user can specify that the program should use this new factory by specifying the system property as follows

```
java -DMyShapeControllerFactory=RandomMyShapeControllerFactory com.deitel.advjhtp1.drawing.DeitelDrawing
```

If the user does not specify a class name in the `MyShapeControllerFactory` system property, the application uses the default `MyShapeControllerFactory` (line 39).

### 5.6.3 Drag-and-Drop Controller

The Deitel Drawing application supports two types of drag-and-drop operations. First, users can drag and drop certain `MyShapes` within drawings and between drawings in the multiple-document interface. Second, users can drag JPEG images from the host operating system’s file manager and drop them on drawings to add those images to drawings.

There are several objects required to enable drag and drop in a Java application. A drag-and-drop operation begins in a `DragSource`. Static method `getDefaultDragSource` of class `DragSource` returns the `DragSource` for the host platform. A `DragGestureRecognizer` recognizes user gestures that begin drag-and-drop operations, such as pressing the mouse button over an object and dragging that object. When a user makes a drag gesture, the `DragGestureRecognizer` notifies its registered `DragGestureListener`s. The `DragGestureListener` then begins the drag-and-drop operation. The user continues the drag gesture until reaching the `DropTarget`, which is the destination for the drag-and-drop operation. When the user makes a gesture to complete the drag-and-drop operation (e.g., by releasing the mouse button), both the `DropTarget` and `DragSource` are notified that the drag-and-drop operation has completed. The event associated with the drag-and-drop operation’s completion includes information about the success or failure of the drag-and-drop operation and a `Transferable` object containing the data that was transferred.

In the Deitel Drawing application, an instance of class `DragAndDropController` (Fig. 5.19) controls each drag-and-drop operation. Class `DragAndDropController` implements three interfaces to handle drag-and-drop operations—`DragGestureListener`, `DragSourceListener` and `DropTargetListener`. These interfaces enable `DragAndDropController` to recognize drag gestures, `DragSource` events and `DropTarget` events.

---

1 // DragAndDropController.java
2 // DragAndDropController is a controller for handling drag and
3 // drop in DeitelDrawing. DragAndDropController implements

---

Fig. 5.19  `DragAndDropController` for moving `MyShapes` between drawings and adding JPEG images to drawings using drag and drop (part 1 of 8).
// DragGestureListener and DragSourceListener to handle drag
// events and DropTargetListener to handle drop events.
package com.deitel.advjhtp1.drawing.controller;

// Java core packages
import java.util.*;
import java.io.*;
import java.awt.Point;
import java.awt.dnd.*;
import java.awt.datatransfer.*;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;

public class DragAndDropController implements DragGestureListener,
        DragSourceListener, DropTargetListener {

    // model to control
    private DrawingModel drawingModel;
    private boolean dragMode = false;

    // DragAndDropController constructor
    public DragAndDropController( DrawingModel model )
    {
        drawingModel = model;
    }

    // set drag mode
    public void setDragMode( boolean drag )
    {
        dragMode = drag;
    }

    // recognize drag operation beginning (method of interface
    // DragGestureListener)
    public void dragGestureRecognized( DragGestureEvent event )
    {
        // if not in dragMode, ignore drag gesture
        if ( !dragMode )
            return;

        // get Point at which drag began
        Point origin = event.getDragOrigin();

        // get MyShapes from DrawingModel
        List shapes = new ArrayList( drawingModel.getShapes() );

        // find top-most shape that contains drag origin (i.e.,
        // start at end of ListIterator and work backwards)

    }
}

Fig. 5.19 DragAndDropController for moving MyShapes between drawings
and adding JPEG images to drawings using drag and drop (part 2 of 8).
ListIterator shapeIterator = 
    shapes.listIterator( shapes.size() );

while ( shapeIterator.hasPrevious() ) {

    MyShape shape = ( MyShape ) shapeIterator.previous();

    if ( shape.contains( origin ) ) {

        // create TransferableShape for dragging shape
        // from Point origin
        TransferableShape transfer =
            new TransferableShape( shape, origin );

        // start drag operation
        event.startDrag( null, transfer, this );

        break;
    }
} // end while

} // end method dragGestureRecognized

// handle drop events (method of interface DropTargetListener)
public void drop( DropTargetDropEvent event )
{

    // get dropped object
    Transferable transferable = event.getTransferable();

    // get dropped object's DataFlavors
    DataFlavor[] dataFlavors =
        transferable.getTransferDataFlavors();

    // get DropTargetDropEvent location
    Point location = event.getLocation();

    // process drops for supported types
    for ( int i = 0; i < dataFlavors.length; i++ ) {
        DataFlavor dataFlavor = dataFlavors[ i ];

        // handle drop of JPEG images
        if ( dataFlavor.equals(
            DataFlavor.javaFileListFlavor ) ) {

            // accept the drop operation
            event.acceptDrop( DnDConstants.ACTION_COPY );

            // attempt to drop the images and indicate whether
            // drop is complete
            event.dropComplete( 
                dropImages( transferable, location ) );
        }
    }

Fig. 5.19 DragAndDropController for moving MyShapes between drawings
and adding JPEG images to drawings using drag and drop (part 3 of 8).
// handle drop of TransferableShape objects
else if ( dataFlavor.isMimeTypeEqual(
    TransferableShape.MIME_TYPE ) ) {

    // accept drop of TransferableShape
    event.acceptDrop( DnDConstants.ACTION_MOVE );

    // drop TransferableShape into drawing
    dropShape( transferable, location );

    // complete drop operation
    event.dropComplete( true );
}

// reject all other DataFlavors
else
    event.rejectDrop();

} // end for

} // end method drop

// drop JPEG images onto drawing
private boolean dropImages( Transferable transferable,
    Point location )
{
    // boolean indicating successful drop
    boolean success = true;

    // attempt to drop images onto drawing
    try {

        // get list of dropped files
        List fileList =
            ( List ) transferable.getTransferData(  
                DataFlavor.javaFileListFlavor );

        Iterator iterator = fileList.iterator();

        // search for JPEG images
        for ( int i = 1; iterator.hasNext(); i++ ) {
            File file = ( File ) iterator.next();

            // if dropped file is a JPEG image, decode and
            // add MyImage to drawingModel
            if ( fileIsJPEG( file ) ) {

                // create MyImage for given JPEG file
                MyImage image = new MyImage();
                image.setFileName( file.getPath() );

Fig. 5.19  DragAndDropController for moving MyShapes between drawings and adding JPEG images to drawings using drag and drop (part 4 of 8).
image.setPoint1( location.x, location.y );

// add to DrawingModel
drawingModel.addShape( image );

else
    success = false;

} // end for

} // end try

// handle exception if DataFlavor not supported
catch ( UnsupportedFlavorException flavorException ) {
    success = false;
    flavorException.printStackTrace();
}

// handle exception reading File
catch ( IOException ioException ) {
    success = false;
    ioException.printStackTrace();
}

return success;

} // end method dropImages

// return true if File has .jpg or .jpeg extension
private boolean fileIsJPEG( File file )
{
    String fileName = file.getName().toLowerCase();

    return fileName.endsWith( ".jpg" ) ||
    fileName.endsWith( ".jpeg" );
}

// drop MyShape object onto drawing
private void dropShape( Transferable transferable,
    Point location )
{
    try {

        DataFlavor flavor = new DataFlavor( ( TransferableShape.MIME_TYPE, "Shape" ) );

        // get TransferableShape object
        TransferableShape transferableShape =
            ( TransferableShape ) transferable.getTransferData( flavor );

        DragAndDropController for moving MyShapes between drawings and adding JPEG images to drawings using drag and drop (part 5 of 8).
// get MyShape and origin Point from TransferableShape
MyShape shape = transferableShape.getShape();
Point origin = transferableShape.getOrigin();

// calculate offset for dropping MyShape
int xOffSet = location.x - origin.x;
int yOffSet = location.y - origin.y;
shape.moveByOffSet( xOffSet, yOffSet );

// add MyShape to target DrawingModel
drawingModel.addShape( shape );

} // end try

// handle exception if DataFlavor not supported
catch ( UnsupportedFlavorException flavorException ) {
    flavorException.printStackTrace();
}

// handle exception getting Transferable data
catch ( IOException ioException ) {
    ioException.printStackTrace();
}

} // end method dropShape

// check for success when drag-and-drop operation ends
// (method of interface DragSourceListener)
public void dragDropEnd( DragSourceDropEvent event ) {
    // if drop successful, remove MyShape from source
    // DrawingModel
    if ( event.getDropSuccess() ) {
        // get Transferable object from DragSourceContext
        Transferable transferable =
            event.getDragSourceContext().getTransferable();

        // get TransferableShape object from Transferable
        try {
            // get TransferableShape object
            TransferableShape transferableShape =
                ( TransferableShape ) transferable.getTransferData( new DataFlavor( TransferableShape.MIME_TYPE, "Shape" ));

            // get MyShape from TransferableShape object
            // and remove from source DrawingModel
drawingModel.removeShape( transferableShape.getShape() );

Fig. 5.19 DragAndDropController for moving MyShapes between drawings and adding JPEG images to drawings using drag and drop (part 6 of 8).
// handle exception if DataFlavor not supported
catch ( UnsupportedFlavorException flavorException ) {
    flavorException.printStackTrace();
}

// handle exception getting transfer data
catch ( IOException ioException ) {
    ioException.printStackTrace();
}

} // end if

} // end method dragDropEnd

// required methods of interface DropTargetListener
public void dragEnter( DropTargetDragEvent event ) {}
public void dragExit( DropTargetEvent event ) {}
public void dragOver( DropTargetDragEvent event ) {}
public void dropActionChanged( DropTargetDragEvent event ) {}

// required methods of interface DragSourceListener
public void dragEnter( DragSourceDragEvent event ) {}
public void dragExit( DragSourceEvent event ) {}
public void dragOver( DragSourceDragEvent event ) {}
public void dropActionChanged( DragSourceDragEvent event ) {}

Fig. 5.19 DragAndDropController for moving MyShapes between drawings and adding JPEG images to drawings using drag and drop (part 7 of 8).
The drag-and-drop subsystem invokes method `dragGestureRecognized` (lines 41–77) when the user makes a drag gesture, such as pressing the mouse button and dragging the mouse on a draggable object. If the user has not selected drag mode in the Deitel Drawing application, line 45 returns to ignore the drag gesture. Lines 58–75 search through the `Collection` of `MyShapes` in the `DrawingModel` for the topmost shape that intersects `Point origin`, which is where the drag gesture occurred. Note that lines 58–75 go through the `Collection` in reverse order, since the topmost shape is at the end of the `Collection`. Lines 66–67 create a `TransferableShape` (Fig. 5.22) that contains the `MyShape` to be dragged and the `Point` at which the drag began. Line 70 invokes method `startDrag` of class `DragGestureEvent` to begin the drag-and-drop operation.

When the user drops a dragged object, the `DropTarget` notifies its `DropTargetListener` by invoking method `drop` (lines 80–129). Line 83 gets the `Transferable` object from the `DropTargetEvent`. Each `Transferable` object contains an array of `DataFlavor`s that describe the type of data contained in the `Transferable` object. Lines 93–127 process the array of `DataFlavor`s to determine the type of object that the user dropped. If the `DataFlavor` is `DataFlavor.javaFileListFlavor` (lines 97–98), the user dropped a `List` of `Files` from the host operating system’s file manager. Line 101 accepts the drop, and line 106 invokes method `dropImages` to process the `File List`. The Deitel Drawing application allows the user to drop only JPEG images, not other file types. If method `dropImages` returns `true`, the files were all JPEG images and the drag-and-drop operation completes successfully. If the `DataFlavor`’s MIME type matches class `TransferableShape`’s MIME type (lines 110–111), line 114 invokes method `acceptDrop` of class `DropTargetDropEvent` to accept the
drop and line 117 invokes method \texttt{dropShape} to process the \texttt{MyShape} that the user dropped. Line 120 invokes method \texttt{dropComplete} of class \texttt{DropTargetDropEvent} to indicate that the drop completed successfully. If the \texttt{DataFlavor} was neither a file list from the file manager nor a \texttt{TransferableShape}, line 125 rejects the drop by invoking method \texttt{rejectDrop} of class \texttt{DropTargetDropEvent}.

Method \texttt{dropImages} (lines 132–186) takes as arguments a \texttt{Transferable} object containing a \texttt{File List} and the \texttt{Point} at which the drop occurred. Lines 142–144 get the \texttt{File List}, and line 146 obtains an \texttt{Iterator} to process the \texttt{List}. Lines 149–168 check each \texttt{File} to determine whether it contains a JPEG image. If the \texttt{File} does contain a JPEG image (line 154), lines 157–162 create a new \texttt{MyImage} object for the JPEG image and add it to the \texttt{DrawingModel}. Method \texttt{fileIsJPEG} (lines 189–195) returns \texttt{true} if the given \texttt{File}’s name ends with the \texttt{.jpg} or \texttt{.jpeg} extension.

Method \texttt{dropShape} (lines 198–253) takes as a \texttt{Transferable} argument the object that the user dropped and a \texttt{Point} argument for the drop location. Lines 207–209 get the \texttt{TransferableShape} object by invoking method \texttt{getTransferData} of interface \texttt{Transferable}. If the \texttt{Transferable} object does not support the \texttt{DataFlavor} passed on line 209, method \texttt{getTransferData} throws an \texttt{UnsupportedFlavorException}. If there is an error reading the data, method \texttt{getTransferData} could throw an \texttt{IOException}. Line 212 gets the \texttt{MyShape} object from the \texttt{TransferableShape}, and line 213 gets the \texttt{Point} from which the \texttt{MyShape} was dragged. Lines 216–219 calculate the offset from the dragged point to the drop point and invoke \texttt{MyShape} method \texttt{moveByOffset} to position the \texttt{MyShape}. Line 222 adds the \texttt{MyShape} to the \texttt{DropTarget}’s \texttt{DrawingModel}.

The drag-and-drop subsystem invokes method \texttt{dragDropEnd} (lines 240–277) when the drag-and-drop operation completes. If the drag-and-drop operation succeeded (line 244), lines 261–262 remove the dragged \texttt{MyShape} object from the source \texttt{DrawingModel}. The remaining empty methods (lines 280–289) satisfy interfaces \texttt{DropTargetListener} and \texttt{DragSourceListener}. Figure 5.20 and Fig. 5.21 describe the methods of interfaces \texttt{DragSourceListener} and \texttt{DropTargetListener}.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{public void dragEnter( DragSourceDragEvent event )}</td>
<td>Invoked when drag-and-drop operation enters the \texttt{DragSource}.</td>
</tr>
<tr>
<td>\texttt{public void dragExit( DragSourceDragEvent event )}</td>
<td>Invoked when drag-and-drop operation exits the \texttt{DragSource}.</td>
</tr>
<tr>
<td>\texttt{public void dragOver( DragSourceDragEvent event )}</td>
<td>Invoked when drag-and-drop operation moves over \texttt{DragSource}.</td>
</tr>
<tr>
<td>\texttt{public void dragDropEnd( DragSourceDragEvent event )}</td>
<td>Invoked when drag-and-drop operation ends.</td>
</tr>
<tr>
<td>\texttt{public void dragDropActionChanged( DragSourceDragEvent event )}</td>
<td>Invoked if user changes drag-and-drop operation (e.g., from copy to move).</td>
</tr>
</tbody>
</table>

\textbf{Fig. 5.20} \texttt{DragSourceListener} interface methods and their descriptions.
Class `TransferableShape` (Fig. 5.22) implements interface `Transferable` to provide a means by which to transfer `MyShape` objects using drag and drop. Interface `Transferable` is part of Java’s data transfer API, which enables clipboard and drag-and-drop functionality in Java applications. Deitel Drawing enables users to drag and drop `TransferableShape` objects between drawings in the multiple-document interface. Lines 24–25 define `static String` variable `MIME_TYPE`, which the data transfer API uses to determine the type of data being transferred. `MIME types (Multipurpose Internet Mail Extension types)` are text strings that were originally created to describe data contained in e-mail attachments sent over the Internet. Many applications and operating systems now use MIME types for the more general purpose of describing objects that contain data, such as files, items on the system clipboard and drag-and-drop objects. Lines 28–29 declare an array of `DataFlavor` objects that class `TransferableShape` supports.

**Fig. 5.21** DropTargetListener interface methods and their descriptions.

```
public void dragEnter( DropTargetDragEvent event )
    Invoked when drag-and-drop operation enters the DropTarget.

public void dragExit( DropTargetEvent event )
    Invoked when drag-and-drop operation exits the DropTarget.

public void dragOver( DropTargetDragEvent event )
    Invoked when drag-and-drop operation moves over DropTarget.

public void drop( DragTargetDropEvent event )
    Invoked when user drops dragged object on DropTarget.

public void dragDropActionChanged( DragSourceDragEvent event )
    Invoked if user changes drag-and-drop operation (e.g., from copy to move).
```

**Fig. 5.22** `TransferableShape` enables DragAndDropController to transfer `MyShape` objects through drag-and-drop operations (part 1 of 3).
public class TransferableShape implements Transferable {

    // the MyShape to transfer from Point origin
    private MyShape shape;
    private Point origin;

    // MIME type that identifies dragged MyShapes
    public static final String MIME_TYPE = 
        "application/x-deitel-shape";

    // DataFlavors that MyShape supports for drag and drop
    private static final DataFlavor[] flavors = new DataFlavor[] {
        new DataFlavor( MIME_TYPE, "Shape" ) };

    // TransferableShape constructor
    public TransferableShape( MyShape myShape, Point originPoint )
    {
        shape = myShape;
        origin = originPoint;
    } // end TransferableShape constructor

    // get Point from which user dragged MyShape
    public Point getOrigin()
    {
        return origin;
    }

    // get MyShape
    public MyShape getShape()
    {
        return shape;
    }

    // get data flavors MyShape supports
    public DataFlavor[] getTransferDataFlavors()
    {
        return flavors;
    }

    // determine if MyShape supports given data flavor
    public boolean isDataFlavorSupported( DataFlavor flavor )
    {
        // search for given DataFlavor in flavors array
        for ( int i = 0; i < flavors.length; i++ )
        {
            if ( flavor.equals( flavors[ i ] ) )
                return true;
        }
        return false;
    }

    Fig. 5.22 TransferableShape enables DragAndDropController to transfer MyShape objects through drag-and-drop operations (part 2 of 3).
Method `getTransferDataFlavors` (lines 52–55) returns the `TransferableShape`’s array of `DataFlavor`s. Method `getTransferDataFlavors` returns a `DataFlavor` array because it is possible that some objects support many `DataFlavor`s. Method `isDataFlavorSupported` (lines 58–67) takes a `DataFlavor` argument that lines 61–64 compare to each `DataFlavor` that class `TransferableShape` supports. If the given `DataFlavor` matches a supported `DataFlavor`, line 64 returns `true`.

Method `getTransferData` (lines 70–78) returns an `Object` containing the data to be transferred by the drag-and-drop operation. The `DataFlavor` argument specifies the particular type of data to be transferred. If the `DataFlavor` argument specifies an invalid `DataFlavor` for class `TransferableShape`, line 74 throws an `UnsupportedFlavorException`. If the `DataFlavor` matches a supported `DataFlavor` for class `MyShape`, line 77 returns a reference to the current `TransferableShape` instance to be transferred.

### 5.7 DrawingInternalFrame Component

`DrawingInternalFrame` (Fig. 5.23) is a `JInternalFrame` subclass that provides a user interface for viewing and modifying drawings. The Deitel Drawing application uses a multiple-document interface to allow the user to view and modify several drawings in a single application window. When the user creates a new drawing or opens a saved drawing, the drawing is displayed in a `DrawingInternalFrame`.

```java
// DrawingInternalFrame.java
// DrawingInternalFrame is a JInternalFrame subclass for
// DeitelDrawing drawings.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.awt. *
import java.awt.event.*;
```
import java.awt.dnd.*;
import java.io.*;
import java.util.*;
import java.util.List;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.model.shapes.*;
import com.deitel.advjhtp1.drawing.view.*;
import com.deitel.advjhtp1.drawing.controller.*;

public class DrawingInternalFrame extends JInternalFrame implements Observer {

    // offsets to stagger new windows
    private static final int xOffset = 30;
    private static final int yOffset = 30;
    private static int openFrameCount = 0;

    // MVC components
    private DrawingModel drawingModel;
    private DrawingView drawingView;
    private MyShapeController myShapeController;
    private DragAndDropController dragAndDropController;
    private MyShapeControllerFactory shapeControllerFactory;

    // file management properties
    private JFileChooser fileChooser;
    private String fileName;
    private String absoluteFilePath;
    private boolean saved = true;

    private DrawingToolBar toolBar;
    private ZoomDialog zoomDialog;

    // Actions for save, zoom, move, etc.
    private Action saveAction, saveAsAction, zoomAction,
    moveAction, fillAction, gradientAction;

    // DrawingInternalFrame constructor
    public DrawingInternalFrame( String title )
    {
         super( title + " - " + ( ++openFrameCount ), true, true,
               false, true );

         setDefaultCloseOperation( WindowConstants.DO_NOTHING_ON_CLOSE );
    }

Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 2 of 15).
// create new DrawingModel
    drawingModel = new DrawingModel();

// create new DrawingView for DrawingModel
drawingView = new DrawingView( drawingModel );

// register DrawingInternalFrame as a DrawingModel Observer
drawingModel.addObserver( this );

// MyShapeControllerFactory for creating MyShapeControllers
shapeControllerFactory =
    MyShapeControllerFactory.getInstance();

// create DragAndDropController for drag and drop operations
dragAndDropController =
    new DragAndDropController( drawingModel );

// get default DragSource for current platform
DragSource dragSource = DragSource.getDefaultDragSource();

// create DragGestureRecognizer to register
// DragAndDropController as DragGestureListener
dragSource.createDefaultDragGestureRecognizer( drawingView,
    DnDConstants.ACTION_COPY_OR_MOVE,
    dragAndDropController );

// enable drawingView to accept drop operations, using
// dragAndDropController as DropTargetListener
drawingView.setDropTarget( new DropTarget( drawingView,
    DnDConstants.ACTION_COPY_OR_MOVE,
    dragAndDropController ) );

// add drawingView to viewPanel, put viewPanel in
// JScrollPane and add JScrollPane to DrawingInternalFrame
JPanel viewPanel = new JPanel();
    viewPanel.add( drawingView );
    getContentPane().add( new JScrollPane( viewPanel ),
        BorderLayout.CENTER );

// create fileChooser and set its FileFilter
fileChooser = new JFileChooser();
    fileChooser.setFileFilter( new DrawingFileFilter() );

// show/hide ZoomDialog when frame activated/deactivated
addInternalFrameListener( new InternalFrameAdapter() {
    // when DrawingInternalFrame activated, make
    // associated zoomDialog visible
    public void internalFrameActivated( InternalFrameEvent event )
      
});
if ( zoomDialog != null )
zoomDialog.setVisible( true );

// when DrawingInternalFrame is deactivated, make
// associated zoomDialog invisible
public void internalFrameDeactivated(
    InternalFrameEvent event )
{
    if ( zoomDialog != null )
        zoomDialog.setVisible( false );
}

// stagger each DrawingInternalFrame to prevent it from
// obscuring other InternalFrames
setLocation( xOffset * openFrameCount,
            yOffset * openFrameCount );

// add new DrawingToolBar to NORTH area
toolBar = new DrawingToolBar();
getContentPane().add( toolBar, BorderLayout.NORTH );

// get name of first MyShape that shapeControllerFactory
// supports and create MyShapeController
String shapeName =
    shapeControllerFactory.getSupportedShapes()[ 0 ];

setMyShapeController(
    shapeControllerFactory.newMyShapeController(
        drawingModel, shapeName ) );

// set DrawingInternalFrame size
setSize( 500, 320 );

// get DrawingInternalFrame Save Action
public Action getSaveAction()
{
    return saveAction;
}

// get DrawingInternalFrame Save As Action
public Action getSaveAsAction()
{
    return saveAsAction;
}

DrawingInternalFrame class that provides a user interface for
creating drawings (part 4 of 15).
165 // set Saved flag for current drawing and update frame
166 // title to indicate saved state to user
167 public void setSaved( boolean drawingSaved )
168 {
169     // set Saved property
170     saved = drawingSaved;
171
172     // get current DrawingInternalFrame title
173     String title = getTitle();
174
175     // if drawing is not saved and title does not end with
176     // an asterisk, add asterisk to title
177     if ( !title.endsWith( " **" ) && !isSaved() )
178         setTitle( title + " **" );
179
180     // if title ends with * and drawing has been saved,
181     // remove * from title
182     else
183         if ( title.endsWith( " **" ) && isSaved() )
184             setTitle( title.substring( 0,
185                 title.length() - 2 ) );
186
187     // enable save actions if drawing not saved
188     getSaveAction().setEnabled( !isSaved() );
189 }
190
191 // return value of saved property
192 public boolean isSaved()
193 {
194     return saved;
195 }
196
197 // handle updates from DrawingModel
198 public void update( Observable observable, Object object )
199 {
200     // set saved property to false to indicate that
201     // DrawingModel has changed
202     setSaved( false );
203 }
204
205 // set fileName for current drawing
206 public void setFileName( String file )
207 {
208     fileName = file;
209
210     // update DrawingInternalFrame title
211     setTitle( fileName );
212 }
213
214 // get fileName for current drawing
215
Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 5 of 15).
216    public String getFileName()
217    {
218        return fileName;
219    }
220
221    // get full path (absoluteFilePath) for current drawing
222    public String getAbsoluteFilePath()
223    {
224        return absoluteFilePath;
225    }
226
227    // set full path (absoluteFilePath) for current drawing
228    public void setAbsoluteFilePath( String path )
229    {
230        absoluteFilePath = path;
231    }
232
233    // get DrawingModel for current drawing
234    public DrawingModel getModel()
235    {
236        return drawingModel;
237    }
238
239    // set JInternalFrame and ZoomDialog titles
240    public void setTitle( String title )
241    {
242        super.setTitle( title );
243
244        if ( zoomDialog != null )
245            zoomDialog.setTitle( title );
246    }
247
248    // set MyShapeController for handling user input
249    public void setMyShapeController( 
250        MyShapeController controller )
251    {
252        // remove old MyShapeController
253        if ( myShapeController != null ) { 
254
255            // remove mouse listeners
256            drawingView.removeMouseListener( 
257                myShapeController.getMouseListener() );
258
259            drawingView.removeMouseMotionListener( 
260                myShapeController.getMouseMotionListener() );
261        }
262
263        // set MyShapeController property
264        myShapeController = controller;
265
266        // register MyShapeController to handle mouse events
267        drawingView.addMouseListener(" Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 6 of 15)."
```java
myShapeController.getMouseListener();

drawingView.addMouseListener(
    myShapeController.getMouseListener();
)

// update new MyShapeController with currently selected
// drawing properties (stroke size, color, fill, etc.)
myShapeController.setStrokeSize( toolBar.getStrokeSize() );

myShapeController.setPrimaryColor( 
    toolBar.getPrimaryColor() );

myShapeController.setSecondaryColor( 
    toolBar.getSecondaryColor() );

myShapeController.setDragMode( toolBar.getDragMode() );

myShapeController.setShapeFilled( 
    toolBar.getShapeFilled() );

myShapeController.setUseGradient( 
    toolBar.getUseGradient() );

} // end method setMyShapeController

// close DrawingInternalFrame; return false if drawing
// was not saved and user canceled the close operation
public boolean close()
{
    // if drawing not saved, prompt user to save
    if ( !isSaved() ) {
        // display JOptionPane confirmation dialog to allow
        // user to save drawing
        int response = JOptionPane.showInternalConfirmDialog( 
            this, "The drawing in this window has been " +
            "modified. Would you like to save changes?",
            "Save Changes", JOptionPane.YES_NO_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE );

        // if user selects Yes, save drawing and close
        if ( response == JOptionPane.YES_OPTION ) {
            saveDrawing();
            dispose();
            return true;
        }

        // if user selects No, close frame without saving
        else if ( response == JOptionPane.NO_OPTION ) {
            dispose();
        }
    }
    return false;
}
```

Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 7 of 15).
```java
    return true;
  }

  // if user selects Cancel, do not save or close
  else
    return false; // indicate frame was not closed

  // if drawing has been saved, close frame
  else {
    dispose();
    return true;
  }

  } // end method close

  // open existing drawing from file
  public boolean openDrawing() {
    // open JFileChooser Open dialog
    int response = fileChooser.showOpenDialog( this ) ;

    // if user selected valid file, open an InputStream
    // and retrieve the saved shapes
    if ( response == fileChooser.APPROVE_OPTION ) {

      // get selecte file name
      String fileName =
        fileChooser.getSelectedFile().getAbsolutePath();

      // get shapes List from file
      Collection shapes =
        DrawingFileReaderWriter.readFile( fileName );

      // set shapes in DrawingModel
      drawingModel.setShapes( shapes );

      // set fileName property
      setFileName( fileChooser.getSelectedFile().getName() );

      // set absoluteFilePath property
      setAbsoluteFilePath( fileName );

      // set saved property
      setSaved( true );

      // return true to indicate successful file open
      return true;
    }

    // return false to indicate file open failed
    else
      return false;
  }
```

Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 8 of 15).
```java
    return false;
}
    // end method openDrawing
    // save current drawing to file
    public void saveDrawing()
    {
        // get absolute path to which file should be saved
        String fileName = getAbsolutePath();
        // if fileName is null or empty, call saveDrawingAs
        if ( fileName == null || fileName.equals( "" ) )
            saveDrawingAs();
        // write drawing to given fileName
        else {
            DrawingFileReaderWriter.writeFile( drawingModel, fileName );
            // update saved property
            setSaved( true );
        }
    } // end method saveDrawing
    // prompt user for file name and save drawing
    public void saveDrawingAs()
    {
        // display JFileChooser Save dialog
        int response = fileChooser.showSaveDialog( this );
        // if user selected a file, save drawing
        if ( response == fileChooser.APPROVE_OPTION )
        {
            // set absoluteFilePath property
            setAbsolutePath( fileChooser.getSelectedFile().getAbsolutePath() );
            // set fileName property
            setFileName( fileChooser.getSelectedFile().getName() );
            // write drawing to file
            DrawingFileReaderWriter.writeFile( drawingModel, getAbsolutePath() );
            // update saved property
            setSaved( true );
        }
    } // end method saveDrawingAs
```
public void showZoomDialog()
{
    // if zoomDialog is null, create one
    if ( zoomDialog == null )
        zoomDialog = new ZoomDialog( getModel(), getTitle() );

    // make extant zoomDialog visible
    else
        zoomDialog.setVisible( true );
}

// JToolBar subclass for DrawingInternalFrame
private class DrawingToolBar extends JToolBar {

    // user interface components
    private GradientIcon gradientIcon;
    private JPanel primaryColorPanel, secondaryColorPanel;
    private JButton primaryColorButton;
    private JButton secondaryColorButton;
    private JComboBox shapeChoice, strokeSizeChoice;
    private JToggleButton gradientButton, fillButton;
    private JToggleButton moveButton;

    // DrawingToolBar constructor
    public DrawingToolBar()
    {
        // create JComboBox for choosing current shape type
        shapeChoice = new JComboBox(
            shapeControllerFactory.getSupportedShapes() );
        shapeChoice.setToolTipText( "Choose Shape" );

        // when shapeChoice changes, get new MyShapeController
        // from MyShapeControllerFactory
        shapeChoice.addActionListener(
            new ActionListener() {

                public void actionPerformed( ActionEvent event )
                {
                    // get selected shape type
                    String className =
                        shapeChoice.getSelectedItem().toString();
setMyShapeController(
    shapeControllerFactory.newMyShapeController(
        drawingModel, className ) );
}

}); // end call to addActionListener

// create JComboBox for selecting stroke size
strokeSizeChoice = new JComboBox(
    new String[] { "1.0", "2.0", "3.0", "4.0", "5.0",
                  "6.0", "7.0", "8.0", "9.0", "10.0" } );

strokeSizeChoice.setToolTipText( "Choose Line Width" );

// set stroke size property to selected value
strokeSizeChoice.addActionListener( 
    new ActionListener() { 
        public void actionPerformed( ActionEvent event ) 
        { 
            myShapeController.setStrokeSize(
                getStrokeSize() );
        }
    });

// create JToggleButton for filling shapes
fillButton = new JToggleButton( "Fill" );

fillAction = new AbstractDrawingAction( "Fill", null,
    "Fill Shape", new Integer( 'L' ) ) {
    public void actionPerformed( ActionEvent event ) 
    { 
        myShapeController.setShapeFilled(
            getShapeFilled() );
    }
};
fillButton.setAction( fillAction );

// create GradientIcon to display gradient settings
gradientIcon = new GradientIcon( Color.black,
    Color.white );

// create JToggleButton to enable/disable gradients
gradientButton = new JToggleButton( gradientIcon );

gradientAction = new AbstractDrawingAction( 
    gradientIcon, "Use Gradient", new Integer( 'G' ) ) {

public void actionPerformed( ActionEvent event )
{
    myShapeController.setUseGradient(
        getUseGradient() );
}
};

gradientButton.addActionListener( gradientAction );

// create JPanel to display primary drawing color
primaryColorPanel = new JPanel();
primaryColorPanel.setPreferredSize(
    new Dimension( 16, 16 ) );
primaryColorPanel.setOpaque( true );
primaryColorPanel.setBackground( Color.black );

// create JButton for changing color1
primaryColorButton = new JButton();
primaryColorButton.add( primaryColorPanel );

// display JColorChooser for selecting startColor value
primaryColorButton.addActionListener(
    new ActionListener() {

        public void actionPerformed( ActionEvent event )
        {
            Color color = JColorChooser.showDialog(
                DrawingInternalFrame.this, "Select Color",
                primaryColorPanel.getBackground() );

            if ( color != null ) {
                primaryColorPanel.setBackground( color );
                gradientIcon.setStartColor( color );
                myShapeController.setPrimaryColor( color );
            }
        }
    } // end ActionListener inner class
); // end call to addActionListener

// create JPanel to display secondary drawing color
secondaryColorPanel = new JPanel();
secondaryColorPanel.setPreferredSize(
    new Dimension( 16, 16 ) );
secondaryColorPanel.setOpaque( true );
secondaryColorPanel.setBackground( Color.white );

// create JButton for changing secondary color
secondaryColorButton = new JButton();
secondaryColorButton.add( secondaryColorPanel );

Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 12 of 15).
// display JColorChooser for selecting endColor value
secondaryColorButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        Color color = JColorChooser.showDialog(DrawingInternalFrame.this, "Select Color", secondaryColorPanel.getBackground());
        if (color != null) {
            secondaryColorPanel.setBackground(color);
            gradientIcon.setEndColor(color);
            myShapeController.setSecondaryColor(color);
        }
    }
}); // end call to addActionListener

// create Action for saving drawings
Icon saveIcon = new ImageIcon(DrawingInternalFrame.class.getResource("images/save.gif"));
saveAction = new AbstractDrawingAction("Save", saveIcon, "Save Drawing", new Integer('S')) {
    public void actionPerformed(ActionEvent event) {
        saveDrawing();
    }
};

// create action for saving drawings as given file name
Icon saveAsIcon = new ImageIcon(DrawingInternalFrame.class.getResource("images/saveAs.gif"));
saveAsAction = new AbstractDrawingAction("Save As", saveAsIcon, "Save Drawing As", new Integer('A')) {
    public void actionPerformed(ActionEvent event) {
        saveDrawingAs();
    }
};

// create action for displaying zoomDialog
Icon zoomIcon = new ImageIcon(Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 13 of 15).
DrawingInternalFrame.class.getResource("images/zoom.gif")

drawAction = new AbstractDrawingAction("Draw", drawIcon, "Draw Shape", null) {
  public void actionPerformed(ActionEvent event) {
    draw();
    } // disable floating
}

moveButton = new JToggleButton();

Icon moveIcon = new ImageIcon(
  DrawingInternalFrame.class.getResource("images/move.gif")
);

moveAction = new AbstractDrawingAction("Move", null, 
  "Move Shape", new Integer('M') ) {
  public void actionPerformed(ActionEvent event) {
    myShapeController.setDragMode(getDragMode());
    dragAndDropController.setDragMode(getDragMode());
  }
}

moveButton.setAction(moveAction);

// add Actions, buttons, etc. to JToolBar
add(saveAction);
add(saveAsAction);
addSeparator();
add(zoomAction);
addSeparator();
add(shapeChoice);
add(strokeSizeChoice);
addSeparator();
add(primaryColorButton);
add(secondaryColorButton);
addSeparator();
add(gradientButton);
add(fillButton);
addSeparator();
add(moveButton);

// disable floating

Fig. 5.23 DrawingInternalFrame class that provides a user interface for creating drawings (part 14 of 15).
Each DrawingInternalFrame has a DrawingModel, DrawingView, MyShapeController and DragAndDropController (lines 36–39). These objects implement the model-view-controller architecture in the Deitel Drawing application. The DrawingInternalFrame’s DrawingView displays the drawing contained in the associated DrawingModel.
Lines 76–92 enable drag and drop in the **DrawingInternalFrame**. Lines 76–77 create a **DragAndDropController**, which controls drag-and-drop operations. Line 80 invokes static method **getDefaultDragSource** of class **DragSource** to get the host platform’s default **DragSource** object. Lines 84–86 invoke method **createDefaultDragGestureRecognizer** of class **DragSource** to register the **DragAndDropController** as the listener for drag gestures that occur inside the **DrawingView**. This enables the **DragAndDropController** to recognize user gestures to drag **MyShapes** in a drawing. Lines 90–92 invoke method **setDropTarget** of class **DrawingView** to enable the **DrawingView** to accept dropped objects, such as **TransferableShapes** from other drawings or JPEG images from the host operating system’s file manager.

Each **DrawingInternalFrame** has an associated **ZoomDialog** that displays a scaled view of the **DrawingModel**. Lines 106–128 create an **InternalFrameListener** that makes the **ZoomDialog** visible when the **DrawingInternalFrame** is activated (lines 111–116) and hides the **ZoomDialog** when the **DrawingInternalFrame** is deactivated (lines 120–125). This ensures that the proper **ZoomDialog** will be displayed when the user switches between **DrawingInternalFrames** in the multiple-document interface.

Recall that the Deitel Drawing application uses a **MyShapeControllerFactory** to create **MyShapeControllers** that handle user input. Lines 141–146 set a **MyShapeController** for the **DrawingInternalFrame**. Line 142 invokes method **getSupportedShapes** of class **MyShapeControllerFactory** to determine for which types of shapes the **MyShapeControllerFactory** can provide **MyShapeControllers**. Lines 141–142 assign the zeroth supported shape to reference **shapeName**. Lines 144–146 invoke **MyShapeControllerFactory** method **newMyShapeController** to obtain an appropriate **MyShapeController** for **shapeName**, and set the **DrawingInternalFrame**’s **MyShapeController**.

Class **DrawingInternalFrame** implements interface **Observer**, so the **DrawingModel** can notify the **DrawingInternalFrame** of changes. When **DrawingInternalFrame** receives an update indicating the **DrawingModel** has changed, method **update** (lines 199–204) invokes method **setSaved** (lines 167–190) with argument **false**. If property **saved** is **false**, method **setSaved** adds an asterisk to the **DrawingInternalFrame**’s title to give the user a visual cue that indicates that the drawing has been modified. If property **saved** is **true**—which indicates that the drawing has not been modified since it was last saved—lines 185–186 remove the asterisk from the **DrawingInternalFrame**’s title. If the drawing has been saved, line 189 disables the **saveAction**. When the drawing is modified, line 189 enables the **saveAction**, which allows the user to save the drawing.

Method **setMyShapeController** (lines 249–291) sets the **MyShapeController** object for controlling mouse input. Lines 253–261 remove the previous **MouseListener** and **MouseMotionListener** from the **DrawingView**. Lines 264–289 register the new **MyShapeController**’s **MouseListener** and **MouseMotionListener** and configure the **MyShapeController** with the currently selected stroke size, colors, drag mode, fill and gradient.

Method **close** (lines 295–334) closes the **DrawingInternalFrame** and prompts the user to save the drawing if the **DrawingModel** has been modified since the last save. Lines 302–306 prompt the user to save an unsaved drawing. If the user selects **Yes**, lines
310–314 save the drawing, invoke method **dispose** to close the **DrawingInternalFrame** and return **true** to indicate that the frame closed successfully. If the user selects **No**, line 319 invokes method **dispose** without saving the drawing, and returns **true** to indicate that the frame closed successfully. If the user selects **Cancel**, the drawing is not saved and line 325 returns **false** to indicate that the **DrawingInternalFrame** was not closed. If the drawing has not been modified since the last save, lines 330–331 invoke method **dispose** and return **true**.

Method **openDrawing** (lines 337–374) opens an existing drawing from the file system. Line 340 displays a **JFileChooser** open dialog. Lines 347–352 get the selected file name and invoke static method **readFile** of class **DrawingFileReaderWriter** to read the **Collection** of shapes from the file.

Method **saveDrawing** (lines 377–395) saves the current drawing. If the drawing does not have an associated file name, line 384 invokes method **saveDrawingAs**. If the drawing does have an associated file name, lines 388–389 invoke static method **writeFile** of class **DrawingFileReaderWriter** to save the drawing. Method **saveDrawingAs** (lines 398–421) displays a **JFileChooser** save dialog to prompt the user for a file in which to save the drawing. Lines 414–415 invoke static method **writeFile** of class **DrawingFileReaderWriter** to save the drawing.

Inner class **DrawingToolBar** (lines 446–726) provides GUI components for saving drawings, selecting the **MyShape** type to draw, and modifying the current **MyShapeController**’s properties, including the stroke size, colors, drag mode, fill mode and gradient. Lines 461–462 populate the **shapeChoice JComboBox** with the array of shape types that the **MyShapeControllerFactory** supports. When the user selects a shape type from **shapeChoice**, lines 477–478 invoke **MyShapeControllerFactory** method **newMyShapeController** to obtain an appropriate **MyShapeController** for the specified shape type. Lines 476–478 invoke method **setMyShapeController** to specify the **MyShapeController** with which the **DrawingInternalFrame** should process user input.

When the user changes the **DrawingInternalFrame**’s state by selecting a new type of shape to draw, that change in state also causes a change in the **DrawingInternalFrame**’s behavior. For example, if the user changes the **MyShapeController** reference from an instance of class **MyTextController** to and instance of class **MyLineController**, the **DrawingInternalFrame** no longer behaves in the same way. Now when the user presses the mouse button and drags the mouse, the **DrawingInternalFrame** draws lines instead of text. This is an example of the **State design pattern**, which enables an object to change its behavior when that object’s state changes. Using the State design pattern becomes beneficial when we add new states (i.e., **MyShapeController** subclasses) to our system—we create an additional **MyShapeController** subclass (e.g., **RandomMyShapeController** in Exercise 5.8) that encapsulates **DrawingInternalFrame**’s behavior when occupying that state.

Lines 485–501 create a **JComboBox** for selecting the stroke size for shapes. Lines 504–516 create a **JToggleButton** and **Action** for creating filled shapes. Lines 519–535 create a **JToggleButton** and **Action** for creating shapes that use gradients. Lines 538–600 create **JButtons** for selecting the **startColor** and **endColor** for shapes. Each **JButton** contains a **JPanel** that displays the currently selected color. When clicked, the **ActionListener** for each **JButton** displays a **JColorChooser** dialog.
that enables the user to select a new color. Lines 603–642 create **Actions** for saving drawings and displaying a **DrawingInternalFrame**’s associated **ZoomDialog** (Fig. 5.25). Lines 645–662 create a **JToggleButton** and **Action** for enabling drag mode, which allows users to drag shapes from this **DrawingInternalFrame**’s **DrawingView**.

Class **DrawingFileFilter** (Fig. 5.24) is a **FileFilter** implementation that enables a **JFileChooser** dialog to show only those files appropriate for our application. Line 15 specifies a description for Deitel Drawing files. Line 8 specifies the filename extension for Deitel Drawing files. Method **accept** (lines 27–31) returns true only if the given **File** matches the filename extension for Deitel Drawing files.

### 5.8 ZoomDialog, Action and Icon Components

Class **ZoomDialog** (Fig. 5.25) is a **JDialog** subclass that uses class **ZoomDrawingView** (Fig. 5.13) to present a scalable **DrawingModel** view. As the user resizes the **ZoomDialog**, the **ZoomDrawingView** adjusts its scale factors to scale to the appropriate size. Note that **ZoomDialog** is a **non-modal dialog** (i.e., the user does not need to close the dialog to continue working with the main portion of the application).

```java
// DrawingFileFilter.java
// DrawingFileFilter is a FileFilter subclass for selecting DeitelDrawing files in a JFileChooser dialog.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.io.File;

// Java extension packages
import javax.swing.filechooser.*;

public class DrawingFileFilter extends FileFilter {

    // String to use in JFileChooser description
    private String DESCRIPTION = "DeitelDrawing Files (*.dd)";

    // file extensions for DeitelDrawing files
    private String EXTENSION = ".dd";

    // get description for DeitelDrawing files
    public String getDescription()
    {
        return DESCRIPTION;
    }

    // return true if given File has proper extension
    public boolean accept( File file )
    {
        return ( file.getName().toLowerCase().endsWith( 
            EXTENSION ) );
    }
}
```

**Fig. 5.24** **DrawingFileFilter** is a **FileFilter** subclass that enables users to select Deitel Drawing files from **JFileChooser** dialogs (part 1 of 2).
Chapter 5  Case Study: Java 2D GUI Application with Design Patterns  305

Fig. 5.25  *ZoomDialog.java* is a *Dialog* subclass that shows a zoomed view of a *DrawingModel*.

```
// ZoomDialog.java
// ZoomDialog is a JDialog subclass that shows a zoomed view of a DrawingModel.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.drawing.model.*;
import com.deitel.advjhtp1.drawing.view.*;

public class ZoomDialog extends JDialog {
    private ZoomDrawingView drawingView;
    private double zoomFactor = 0.5;

    // ZoomDialog constructor
    public ZoomDialog( DrawingModel model, String title )
    {
        // set ZoomDialog title
        setTitle( title);

        // create ZoomDrawingView for using default zoomFactor
        drawingView = new ZoomDrawingView( model, zoomFactor );

        // add ZoomDrawingView to ContentPane
        getContentPane().add( drawingView );

        // size ZoomDialog to fit ZoomDrawingView's preferred size
        pack();

        // make ZoomDialog visible
        setVisible( true );
    }

    // set JDialog title
    public void setTitle( String title )
    {
        super.setTitle( title + " [Zoom]");
    }
}
```

Fig. 5.24  *DrawingFileFilter* is a *FileFilter* subclass that enables users to select Deitel Drawing files from *JFileChooser* dialogs (part 2 of 2).
Class AbstractDrawingAction is an abstract base class that extends class AbstractAction to provide a more convenient way to create Swing Actions. The AbstractDrawingAction constructor (lines 16–24) takes as arguments the name, Icon, description and mnemonic for the Action. Lines 27–48 define set methods for each Action property. Method actionPerformed (line 52) is marked abstract to require an implementation in each subclass.

```java
// AbstractDrawingAction.java
// AbstractDrawingAction is an Action implementation that
// provides set and get methods for common Action properties.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public abstract class AbstractDrawingAction
    extends AbstractAction {
    // construct AbstractDrawingAction with given name, icon
    // description and mnemonic key
    public AbstractDrawingAction( String name, Icon icon,
        String description, Integer mnemonic )
    {
        setName( name );
        setSmallIcon( icon );
        setShortDescription( description );
        setMnemonic( mnemonic );
    }

    // set Action name
    public void setName( String name )
    {
        putValue( Action.NAME, name );
    }

    // set Action Icon
    public void setSmallIcon( Icon icon )
    {
        putValue( Action.SMALL_ICON, icon );
    }

    // set Action short description
    public void setShortDescription( String description )
    {
        putValue( Action.SHORT_DESCRIPTION, description );
    }

Fig. 5.26 AbstractDrawingAction abstract base class for Actions
(part 1 of 2).
```
Class \texttt{GradientIcon} implements interface \texttt{Icon} and draws a gradient from \texttt{startColor} (line 15) to \texttt{endColor} (line 16). The Deitel Drawing application uses a \texttt{GradientIcon} to show a preview of the currently selected colors drawn as a gradient. Method \texttt{paintIcon} (lines 62–75) draws a filled rectangle, using a Java2D \texttt{GradientPaint} and the \texttt{GradientIcon}'s \texttt{startColor} and \texttt{endColor}.

### Fig. 5.27 \texttt{GradientIcon} implementation of interface \texttt{Icon} that draws a gradient (part 1 of 2).

```java
// GradientIcon.java
// GradientIcon is an Icon implementation that draws a 16 x 16
// gradient from startColor to endColor.
package com.deitel.advjhtp1.painting;

// Java core packages
import java.awt.*;

// Java extension packages
import javax.swing.*;

public class GradientIcon implements Icon {

    // Colors to use for gradient
    private Color startColor;
    private Color endColor;

    // GradientIcon constructor
    public GradientIcon( Color start, Color end )
    {
        setStartColor( start );
        setEndColor( end );
    }

    // set gradient start color
    public void setStartColor( Color start )
    {
        startColor = start;
    }

    // set Action mnemonic key
    public void setMnemonic( Integer mnemonic )
    {
        putValue( Action.MNEMONIC_KEY, mnemonic );
    }

    // abstract actionPerformed method to be implemented
    // by concrete subclasses
    public abstract void actionPerformed( ActionEvent event );
}
```
```java
// get gradient start color
public Color getStartColor()
{
    return startColor;
}

// set gradient end color
public void setEndColor( Color end )
{
    endColor = end;
}

// get gradient end color
public Color getEndColor()
{
    return endColor;
}

// get icon width
public int getIconWidth()
{
    return 16;
}

// get icon height
public int getIconHeight()
{
    return 16;
}

// draw icon at given location on given component
public void paintIcon( Component component, Graphics g,
int x, int y )
{
    // get Graphics2D object
    Graphics2D g2D = ( Graphics2D ) g;

    // set GradientPaint
    g2D.setPaint ( new GradientPaint( x, y,
        getStartColor(), 16, 16,
        getEndColor() ) );

    // fill rectangle with gradient
    g2D.fillRect( x, y, 16, 16 );
}
```

Fig. 5.27 GradientIcon implementation of interface Icon that draws a gradient (part 2 of 2).
5.9 DeitelDrawing Application

Class DeitelDrawing (Fig. 5.28) integrates the components we have discussed in this chapter into a multiple-document-interface application. The Deitel Drawing application displays the Deitel logo in a SplashScreen (Fig. 5.29) while the application loads (line 44). Lines 60–111 create AbstractDrawingActions for creating new drawings, opening existing drawings, exiting the application and displaying information about the Deitel Drawing application.

```java
// DeitelDrawing.java
// DeitelDrawing is a drawing program that uses, MVC, a
// multiple-document interface and Java2D.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class DeitelDrawing extends JFrame {

    private JMenuBar menuBar;
    private JMenu fileMenu, helpMenu;

    private Action newAction, openAction,
        exitAction, aboutAction;

    private JMenuItem saveMenuItem, saveAsMenuItem;

    private JToolBar toolBar;
    private JPanel toolBarPanel, frameToolBarPanel;
    private JDesktopPane desktopPane;

    private SplashScreen splashScreen;

    // DeitelDrawing constructor
    public DeitelDrawing()
    {
        super( "DeitelDrawing" );

        // set icon for JFrame's upper-left-hand corner
        ImageIcon icon = new ImageIcon(
            DeitelDrawing.class.getResource( "images/icon.png" ) );
        setDefaultCloseOperation( IconImage( icon.getImage() ) );
    }
}
```

Fig. 5.28 DeitelDrawing application that uses a multiple-document interface for displaying and modifying DeitelDrawing drawings (part 1 of 8).
showSplashScreen();

// do not hide window when close button clicked
setDefaultCloseOperation(
    WindowConstants.DO_NOTHING_ON_CLOSE);

// create JDesktopPane for MDI
desktopPane = new JDesktopPane();

// show contents when dragging JInternalFrames
desktopPane.setDragMode(JDesktopPane.LIVE_DRAG_MODE);

// create Action for creating new drawings
Icon newIcon = new ImageIcon(
    DeitelDrawing.class.getResource("images/new.gif"));

newAction = new AbstractDrawingAction("New", newIcon,
    "Create New Drawing", new Integer('N')) {
    public void actionPerformed(ActionEvent event) {
        createNewWindow();
    }
};

// create Action for opening existing drawings
Icon openIcon = new ImageIcon(
    DeitelDrawing.class.getResource("images/open.gif"));

openAction = new AbstractDrawingAction("Open", openIcon,
    "Open Existing Drawing", new Integer('O')) {
    public void actionPerformed(ActionEvent event) {
        DrawingInternalFrame frame = createNewWindow();
        if (!frame.openDrawing())
            frame.close();
    }
};

// create Action for exiting application
Icon exitIcon = new ImageIcon(
    DeitelDrawing.class.getResource("images/exit.gif"));

exitAction = new AbstractDrawingAction("Exit", exitIcon,
    "Exit Application", new Integer('X')) {
    public void actionPerformed(ActionEvent event) {
        exitApplication();
    }
};

Fig. 5.28  DeitelDrawing application that uses a multiple-document interface
for displaying and modifying DeitelDrawing drawings (part 2 of 8).
// create Action for opening About dialog
Icon aboutIcon = new ImageIcon(
    DeitelDrawing.class.getResource( "images/about.gif" ) );

aboutAction = new AbstractDrawingAction( "About",
    aboutIcon, "About Application", new Integer( 'b' ) ) {

    public void actionPerformed( ActionEvent event )
    {
        JOptionPane.showMessageDialog( DeitelDrawing.this,
            "DeitelDrawing v1.0.
            Copyright " +
            "2002. Deitel & Associates, Inc." );
    }
};

// create File menu and set its mnemonic
fileMenu = new JMenu( "File" );
fileMenu.setMnemonic( 'F' );

// create Help menu and set its mnemonic
helpMenu = new JMenu( "Help" );
helpMenu.setMnemonic( 'H' );

menuBar = new JMenuBar();

// add New Drawing and Open Drawing actions to
// File menu and remove their icons
fileMenu.add( newAction ).setIcon( null );
fileMenu.add( openAction ).setIcon( null );

// create JMenuItems for saving drawings; these
// JMenuItems will invoke the save Actions for the
// current DrawingInternalFrame
saveMenuItem = new JMenuItem( "Save" );
saveAsMenuItem = new JMenuItem( "Save As" );

// add Save, Save As and Close JMenuItems to File menu
fileMenu.add( saveMenuItem );
fileMenu.add( saveAsMenuItem );

fileMenu.addSeparator();

// add Exit action to File menu and remove its icon
fileMenu.add( exitAction ).setIcon( null );

// add About action to Help menu and remove its icon
helpMenu.add( aboutAction ).setIcon( null );

// add File and Help menus to JMenuBar

Fig. 5.28 DeitelDrawing application that uses a multiple-document interface for displaying and modifying DeitelDrawing drawings (part 3 of 8).
312  Case Study: Java 2D GUI Application with Design Patterns  Chapter 5

```java
menuBar.add( fileMenu );
menuBar.add( helpMenu );

// set Frame's JMenuBar
setJMenuBar( menuBar );

// create application JToolBar
toolBar = new JToolBar();

// disable JToolBar floating
toolBar.setFloatable( false );

// add New Drawing and Open Drawing actions to JToolBar
toolBar.add( newAction );
toolBar.add( openAction );

toolBar.addSeparator();

// add Exit action to JToolBar
toolBar.add( exitAction );

toolBar.addSeparator();

// add About action to JToolBar
toolBar.add( aboutAction );

// add toolBar and desktopPane to ContentPane
getContentPane().add( toolBar, BorderLayout.NORTH );
getContentPane().add( desktopPane, BorderLayout.CENTER );

// add WindowListener for windowClosing event
addWindowListener(
    new WindowAdapter() {
        public void windowClosing( WindowEvent event )
        {
            exitApplication();
        }
    });

// wait for SplashScreen to go away
while ( splashScreen.isVisible() ) {
    try {
        Thread.sleep( 10 );
    } catch ( InterruptedException interruptedException ) {
        interruptedException.printStackTrace();
    }
}
```

**Fig. 5.28** DeitelDrawing application that uses a multiple-document interface for displaying and modifying DeitelDrawing drawings (part 4 of 8).
// set initial JFrame size
setSize( 640, 480 );

// position application window
centerWindowOnScreen();

// make application visible
setVisible( true );

// create new, empty drawing window
createNewWindow();

} // end DeitelDrawing constructor

// create new DrawingInternalFrame
private DrawingInternalFrame createNewWindow()
{
    // create new DrawingInternalFrame
    DrawingInternalFrame frame =
        new DrawingInternalFrame( "Untitled Drawing" );

    // add listener for InternalFrame events
    frame.addInternalFrameListener(
        new DrawingInternalFrameListener() );

    // make DrawingInternalFrame opaque
    frame.setOpaque( true );

    // add DrawingInternalFrame to desktopPane
    desktopPane.add( frame );

    // make DrawingInternalFrame visible
    frame.setVisible( true );

    // select new DrawingInternalFrame
    try {
        frame.setSelected( true );
    }

    // handle exception selecting DrawingInternalFrame
    catch ( PropertyVetoException vetoException ) {
        vetoException.printStackTrace();
    }

    // return reference to newly created DrawingInternalFrame
    return frame;

} // end createNewWindow

// InternalFrameAdapter to listen for InternalFrame events

Fig. 5.28  DeitelDrawing application that uses a multiple-document interface for displaying and modifying DeitelDrawing drawings (part 5 of 8).
private class DrawingInternalFrameListener
  extends InternalFrameAdapter {

    // when DrawingInternalFrame is closing disable
    // appropriate Actions
    public void internalFrameClosing(
        InternalFrameEvent event )
    {
        DrawingInternalFrame frame =
            ( DrawingInternalFrame ) event.getSource();

        // frame closes successfully, disable Save menu items
        if ( frame.close() ) {
            saveMenuItem.setAction( null );
            saveAsMenuItem.setAction( null );
        }
    }

    // when DrawingInternalFrame is activated, make its JToolBar
    // visible and set JMenuItems to DrawingInternalFrame Actions
    public void internalFrameActivated(
        InternalFrameEvent event )
    {
        DrawingInternalFrame frame =
            ( DrawingInternalFrame ) event.getSource();

        // set saveMenuItem to DrawingInternalFrame's saveAction
        saveMenuItem.setAction( frame.getSaveAction() );
        saveMenuItem.setIcon( null );

        // set saveAsMenuItem to DrawingInternalFrame's
        // saveAsAction
        saveAsMenuItem.setAction( frame.getSaveAsAction() );
        saveAsMenuItem.setIcon( null );
    }

    // close each DrawingInternalFrame to let user save drawings
    // then exit application
    private void exitApplication()
    {
        // get array of JInternalFrames from desktopPane
        JInternalFrame frames[] = desktopPane.getAllFrames();

        // keep track of DrawingInternalFrames that do not close
        boolean allFramesClosed = true;

        // select and close each DrawingInternalFrame
        for ( int i = 0; i < frames.length; i++ )
            if ( frames[i] instanceof DrawingInternalFrame )
                frames[i].close();

        // exit application
        System.exit(0);
    }

Fig. 5.28  DeitelDrawing application that uses a multiple-document interface for displaying and modifying DeitelDrawing drawings (part 6 of 8).
// select current DrawingInternalFrame
try {
    nextFrame.setSelected( true );
}

// handle exception when selecting DrawingInternalFrame
catch ( PropertyVetoException vetoException ) {
    vetoException.printStackTrace();
}

// close DrawingInternalFrame and update allFramesClosed
allFramesClosed = allFramesClosed && nextFrame.close();

// exit application only if all frames were closed
if ( allFramesClosed )
    System.exit( 0 );

} // end method exitApplication

// display application's splash screen
public void showSplashScreen()
{
    // create ImageIcon for logo
    Icon logoIcon = new ImageIcon(
        getClass().getResource( "images/deitellogo.png" ) );

    // create new JLabel for logo
    JLabel logoLabel = new JLabel( logoIcon );

    // set JLabel background color
    logoLabel.setBackground( Color.white );

    // set splash screen border
    logoLabel.setBorder( new MatteBorder( 5, 5, 5, 5, Color.black ) );

    // make logoLabel opaque
    logoLabel.setOpaque( true );

    // create SplashScreen for logo
    splashScreen = new SplashScreen( logoLabel );

    // show SplashScreen for 3 seconds
    splashScreen.showSplash( 3000 );

} // end method showSplashScreen

// center application window on user's screen
private void centerWindowOnScreen()
{
    // get Dimension of user's screen

Fig. 5.28  DeitelDrawing application that uses a multiple-document interface for displaying and modifying DeitelDrawing drawings (part 7 of 8).
Method `createNewWindow` (lines 216–247) creates a new `DrawingInternalFrame`. Inner class `DrawingInternalFrameListener` (lines 250–285) listens for `internalFrameClosing` and `internalFrameActivated` messages. The Deitel Drawing application’s `File` menu contains `JMenuItem` s for saving the currently active drawing. When a `DrawingInternalFrame` closes, lines 263–264 remove that `DrawingInternalFrame`’s `saveAction` and `saveAsAction` from `saveMenuItem` and `saveAsMenuItem`. When a `DrawingInternalFrame` is activated, lines 277–283 invoke method `setAction` of class `JMenuItem` to set the Actions for `saveMenuItem` and `saveAsMenuItem`.

Method `exitApplication` (lines 289–320) prompts the user to save any unsaved drawings before the application exits. Line 292 invokes method `getAllFrames` of class `JDesktopPane` to retrieve an array of `JInternalFrame`s in the application. Line 313 invokes method `close` of class `DrawingInternalFrame` to attempt to close each `DrawingInternalFrame` in the array. Method `close` returns `true` if the `DrawingInternalFrame` closed successfully, `false` otherwise. Line 313 accumulates the results of closing each `DrawingInternalFrame` in boolean `allFramesClosed`. If all `DrawingInternalFrame`s close successfully, line 318 exits the application. If any `DrawingInternalFrame` did not close, the application assumes that the user cancelled the request to close the application.

The Deitel Drawing application displays the Deitel logo in a `SplashScreen` (Fig. 5.29) while the application loads. The `SplashScreen` constructor (lines 19–58) takes as an argument the `Component` to display. Line 22 creates `JWindow` (a borderless window) in which to display the given component. Lines 46–56 center the `SplashScreen`’s `JWindow` on the user’s screen.
// SplashScreen.java
// SplashScreen implements static method showSplash for
// displaying a splash screen.
package com.deitel.advjhtp1.drawing;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class SplashScreen {

private JWindow window;
private Timer timer;

// SplashScreen constructor
public SplashScreen( Component component ) {
    // create new JWindow for splash screen
    window = new JWindow();

    // add provided component to JWindow
    window.getContentPane().add( component );

    // allow user to dismiss SplashScreen by clicking mouse
    window.addMouseListener(
        new MouseAdapter() {
            // when user presses mouse in SplashScreen,
            // hide and dispose JWindow
            public void mousePressed( MouseEvent event ) {
                window.setVisible( false );
                window.dispose();
            }
        }
    ); // end call to addMouseListener

    // size JWindow for given Component
    window.pack();

    // get user's screen size
    Dimension screenDimension =
        Toolkit.getDefaultToolkit().getScreenSize();

    // calculate x and y coordinates to center splash screen
    int width = window.getSize().width;
    int height = window.getSize().height;
    int x = ( screenDimension.width - width ) / 2 ;

    // Fig. 5.29 SplashScreen class for displaying a logo while the application loads (part 1 of 2).
int y = (screenDimension.height - height) / 2;

// set the bounds of the window to center it on screen
window.setBounds(x, y, width, height);

} // end SplashScreen constructor

// show splash screen for given delay
public void showSplash(int delay) {

    // display the window
    window.setVisible(true);

    // create and start a new Timer to remove SplashScreen
    // after specified delay
    timer = new Timer(delay,
    new ActionListener() {

        public void actionPerformed(ActionEvent event) {

            // hide and dispose of window
            window.setVisible(false);
            window.dispose();
            timer.stop();
        }
    });

    timer.start();

} // end method showSplash

// return true if SplashScreen window is visible
public boolean isVisible() {
    return window.isVisible();
}

Fig. 5.29 **SplashScreen** class for displaying a logo while the application loads (part 2 of 2).
Method `showSplash` (lines 61–83) takes as an integer argument the number of milliseconds for which to display the `SplashScreen`. Line 64 makes the `JWindow` visible, and line 51 causes the current `Thread` to sleep for the given `delay`. After the `delay` expires, lines 60–61 hide and `dispose` of the `JWindow`.

In this chapter, we presented a substantial application that used the MVC architecture and many popular design patterns, including Observer, Factory Method, Template Method, State and Command. We also demonstrated how applications can store and retrieve information in XML documents. Our drawing application takes advantage of the rich set of GUI components offered by Swing and the powerful drawing capabilities offered by Java 2D. Drag-and-drop functionality enables users to transfer shapes between drawings and add their own images.

Throughout the rest of the book, we use design patterns and the MVC architecture to build substantial examples and case studies. For example, the Enterprise Java case study of Chapters 17–20 presents an online bookstore that uses the MVC architecture.

**SELF-REVIEW EXERCISES**

5.1 Which part of the model-view-controller architecture processes user input? Which Deitel Drawing classes implement this part of MVC?

5.2 What interface must a class implement to enable data transfer using drag and drop for instances of that class?

5.3 In general, how does a user begin a drag-and-drop operation? Give an example.

5.4 What type of object notifies a `DragGestureListener` that the user made a drag gesture?

5.5 How can a `DragTargetListener` or `DragSourceListener` determine what type of data a `Transferable` object contains?

**ANSWERS TO SELF-REVIEW EXERCISES**

5.1 The controller in MVC processes user input. In the Deitel Drawing application, `MyShapeController` subclasses process user input via the mouse. Class `DragAndDropController` processes user input via drag-and-drop operations.

5.2 A class that supports drag and drop must implement interface `Transferable`.

5.3 A user begins a drag-and-drop operation by making a drag gesture. For example, on the Windows platform, a user makes a drag gesture by pressing the mouse button on a draggable object and dragging the mouse.

5.4 A `DragGestureRecognizer` issues a `DragGestureEvent` to notify a `DragGestureListener` that the user made a drag gesture.

5.5 Method `getTransferDataFlavors` of interface `Transferable` returns an array of `DataFlavor` objects. Each `DataFlavor` has a MIME type that describes the type of data the `Transferable` object supports.

**EXERCISES**

5.6 Create class `RotatingDrawingView` that extends class `DrawingView` and uses Java 2D transformations (Chapter 4) to display the drawing rotated by ninety degrees.

5.7 Modify your solution to Exercise 5.6 to use a and a `java.awt.Timer` to continually rotate the drawing in five-degree increments.
Create class `RandomMyShapeController` that extends class `MyShapeController` and adds random `MyShape` subclasses with random sizes, colors and other properties to the `DrawingModel`. In method `startShape`, class `RandomMyShapeController` should prompt the user for the number of random shapes to add to the drawing. Create a new `MyShapeController-Factory` subclass (Fig. 5.18) named `RandomMyShapeControllerFactory` that constructs a `RandomMyShapeController` when the String "Random" is passed to method `newMyShapeController`. [Hint: Be sure to override method `getSupportedShapes` of class `MyShapeControllerFactory` to return a `String` array that includes the String "Random".]
Objectives

- To understand JavaBeans and how they facilitate component-oriented software construction.
- To be able to use Forte for Java Community Edition to build JavaBeans-based applications.
- To be able to wrap class definitions as JAR files for use as JavaBeans and stand-alone applications.
- To be able to define JavaBean properties and events.

*Mirrors should reflect a little before throwing back images.*
Jean Cocteau

*Television is like the invention of indoor plumbing. It didn’t change people’s habits. It just kept them inside the house.*
Alfred Hitchcock

*The power of the visible is the invisible.*
Marianne Moore

*The sun has a right to "set" where it wants to, and so, I may add, has a hen.*
Charles Farrar Browne

*The causes of events are ever more interesting than the events themselves.*
Marcus Tullius Cicero

*…the mechanic that would perfect his work must first sharpen his tools.*
Confucius
6.1 Introduction

This chapter presents Java’s reusable software component model: *JavaBeans*. JavaBeans (often called *beans*) allow developers to reap the benefits of rapid application development in Java by assembling predefined software components to create powerful applications and applets. *Graphical programming and design environments* (often called *builder tools*, IDEs or *integrated development environments*) that support beans provide programmers with tremendous flexibility by allowing programmers to reuse and integrate existing disparate components that, in many cases, were never intended to be used together. These components can be linked together to create applets, applications or even new beans for reuse by others.

JavaBeans and other component-based technologies have led to a new type of programmer, the *component assembler*, who uses well-defined components to create more robust functionality. Component assemblers do not need to know the implementation details of components. Rather, they need to know what services the components provide, so they can have other components interact with them.

As an example of the concept of beans, assume that a component assembler has an animation bean that has methods to `startAnimation` and `stopAnimation`. The component assembler may want to provide two buttons, one that will start the animation and one that will stop the animation (an example you will see later in this chapter). With beans, we can simply “connect” one button to the animation’s `startAnimation` method and connect another button to the animation’s `stopAnimation` method, such that when the user clicks a button, the appropriate method of the animation bean is called. The builder tool does all the work of associating the button-click event with the appropriate method to call on the animation bean. All the programmer needs to do is tell the builder tool which two components to “connect.”

The benefit of beans in this example is that the animation bean and the button beans do not need to know about each other before they are assembled in a builder tool. Someone else can be responsible for defining the concept of a button in a reusable manner (e.g.,
javax.swing.JButton). A button is not specific to our example. Rather, it is a component used in many applications and applets. When the user of a program clicks a button, the user expects an action specific to that program to occur. (Some buttons, such as OK buttons, typically have the same meaning in all programs.) However, the basic concept of a button—how it is displayed, how it works and how it notifies other components that it was clicked—is the same in every application (although we typically customize the button’s label). The component assembler’s job is not to create the concept of a button, but rather to use the preexisting button component to provide functionality to the user of the program.

Component assemblers can make beans communicate through the beans’ well-defined services (i.e., methods), typically without writing any code (the builder tool often generates the code, which is sometimes hidden from the component assembler—depending on the tool). Indeed, a component assembler often can create complex applications literally by "connecting the dots."

In this chapter, we show you how to use existing beans and how to create your own basic beans. After studying this chapter, you will have a foundation in JavaBeans programming that will enable you to develop applications and applets rapidly using the more advanced features of integrated development environments that support beans. You will also have a solid foundation for further study of JavaBeans.

For more JavaBeans information, visit the Sun Microsystems Web site for JavaBeans:

java.sun.com/beans/

This site provides a complete set of resources for learning about and using JavaBeans.

### 6.2 Using Beans in Forte for Java Community Edition

Sun Microsystems’s *Forte for Java Community Edition* (Fig. 6.1) is an integrated development environment that provides a builder tool for assembling JavaBeans. Forte provides visual access to a variety of JavaBeans and allows you to install and manipulate additional beans. In this section, we demonstrate how to use existing beans in Forte. Later in the chapter, we rely on your knowledge of this section to use the beans created in this chapter. We assume you are already familiar with the basic operation of *Forte*. For details on getting started with *Forte*, visit the resources for this book on our Web site, www.deitel.com. There, we have a “Getting Started with *Forte for Java Community Edition 2.0*” tutorial.

**Software Engineering Observation 6.1**

A benefit of working in a bean-ready development environment is that the environment visually presents the properties of the bean to the programmer for easy modification and customization of the bean at design time.

A bean must be installed before it is manipulated in Forte. Click the **Tools** menu and select **Install New JavaBean...** (Fig. 6.2). A file dialog box labelled **Install JavaBean** appears (Fig. 6.3). Copy **LogoAnimator.jar** from the CD-ROM that accompanies this book. The next dialog box lists the JavaBeans within the selected JAR file (Fig. 6.4). Select **LogoAnimator** and click the **OK** button (Fig. 6.4). Select **Beans** in the **Palette Category** dialog box that appears next and click **OK** (Fig. 6.4). Clicking the **Beans** tab in the **Component Palette** shows a question mark icon (Fig. 6.5). Moving the mouse over the icon in the **Component Palette** displays a tool tips showing that the icon represents the **LogoAnimator** JavaBean (Fig. 6.5).
Fig. 6.1  Forte for Java Community Edition 2.0.

Fig. 6.2  Install New JavaBean... menu item.

Fig. 6.3  Install JavaBean dialog.
GUI JavaBeans must be added to a Java Container to be able to use the builder tool to edit the bean properties or to link the beans to other components. To demonstrate adding and manipulating JavaBeans, we open a JFrame. Select the Filesystems tab in the Explorer window (Fig. 6.6). Select the Development directory (Fig. 6.7). Select New... from the File menu (Fig. 6.8). In the Template Chooser (Fig. 6.9), expand the Swing Forms option and select JFrame. Enter “AnimationWindow” in the Name: field (Fig. 6.9). Click Finish to create the new JFrame.
The new `AnimationWindow` class appears inside the `Filesystems` field of the `Explorer`. The `Component Inspector`, `Form` and `Source Editor` windows should all appear (Fig. 6.10). The `Component Inspector` (Fig. 6.11) lists all the visual and nonvisual components within `AnimationWindow` and also shows the property sheet for selected components (we will discuss the property sheet later). The `Form` window (Fig. 6.11) shows the `JFrame` with its current layout and components. The `Source`
Editor (Fig. 6.12) shows the Java source code Forte generates. Forte updates this code as components and events are added, deleted and changed.
We now begin building the application by placing the LogoAnimator JavaBean we just imported into the AnimationWindow. Click the Beans tab of the Component Palette (Fig. 6.13). Next, click the LogoAnimator icon (Fig. 6.14). Then, click in the Form window in the center of the JFrame. A spinning animation of the Deitel and Associates, Inc., logo will appear in the window (Fig. 6.15).
The property sheet in the Component Inspector displays a component’s properties and allows them to be edited. Click the LogoAnimator in the Form window. Blue squares appear at the corners of the animation to show it is selected (Fig. 6.15). The Component Inspector shows all the LogoAnimator properties (Fig. 6.16). Many of the properties are inherited from JPanel, the superclass of LogoAnimator. The background property shows a swatch of color and a name indicating the LogoAnimator background color. Click the color, and a drop-down menu appears (Fig. 6.17). It lists some of the predefined colors in Java. Select the first color listed in the drop-down menu to change the LogoAnimator’s background to white (Fig. 6.18). Try selecting other colors to get used to changing JavaBean properties.

Fig. 6.15  LogoAnimator animation in the Form window.

Fig. 6.16  Component Inspector with LogoAnimator Properties sheet.
In addition to changing JavaBean properties with the builder tool, component assem-blers can connect JavaBeans with events. For instance, a button can control the function of another component. We demonstrate this with buttons that start and stop the LogoAni-
mator’s animation.

**Fig. 6.17** Component Inspector drop down-menu for the **background** property.

**Fig. 6.18** Changing **background** color of LogoAnimator.
Before adding other components to our example, we change the window’s layout to a **FlowLayout**. In the **Explorer** window, expand the **AnimationWindow** node (Fig. 6.19). Right click the **JFrame** node, select **Set Layout** and click **FlowLayout** (Fig. 6.20).

Select the **Swing** tab in the **Component Palette** (Fig. 6.21). This tab contains the most common Swing components. The second component in the list is the **JButton** (Fig. 6.22). Click the **JButton** icon, then click an empty spot in the **Form** that contains the **LogoAnimator**. A new **JButton** appears in the window next to the **LogoAnimator** (Fig. 6.23). Select the **JButton** and locate the **text** property in the **Component Inspector**. Click the text field, type **Start Animation** (Fig. 6.24), then press **Enter**. The button text in the **Form** will change to the new value (Fig. 6.24). Repeat this procedure to add another **JButton** with the text **Stop Animation**.

---

**Fig. 6.19** **AnimationWindow** selected in **Explorer**.

**Fig. 6.20** Selecting **FlowLayout** in the **Explorer** menu.
Fig. 6.21  Swing tab of the Component Palette.

Fig. 6.22  JButton icon in the Component Palette.

Fig. 6.23  Adding a JButton to AnimationWindow.

Fig. 6.24  Editing text property of JButton.
Next, we connect the **Start Animation** and **Stop Animation** buttons to the **Logo-Animator** so the user can start and stop the animation. The button with the mouse pointer icon to the left of the **Component Palette** enables **Selection Mode** (Fig. 6.25). This mode enables Forte users to select components in a **Form** window. The button with the double-arrows icon below the **Selection Mode** icon enables **Connection Mode** (Fig. 6.26), which allows Forte users to connect components with a wizard that generates code in the **Source Editor**. Click the **Connection Mode** icon to enter **Connection Mode** (Fig. 6.27). Click the **Start Animation JButton** (Fig. 6.28), which will be the source of the event (i.e., the source component) that starts the animation. Red squares appear at the corners of the **JButton**. Next, click the **LogoAnimator**. Red squares also appear at the corners of **LogoAnimator** and the **Connection Wizard** dialog appears (Fig. 6.29).

Step 1 of the **Connection Wizard** lists all the events that the source component supports. In this application, we want the button click event to call the animator’s **startAnimation** method, so we need to connect the button’s action event to **LogoAnimator**’s **start-Animation** method. Expand the **action** node, highlight **actionPerformed** and click the **Next** button at the bottom of the **Connection Wizard** (Fig. 6.30). Step 2 (Fig. 6.31) lists the methods or properties that can be set on the target component (**LogoAnimator**). Click the **Method Call** radio button to show a list of **LogoAnimator**’s methods. Many of the methods that appear in the list are inherited from **LogoAnimator**’s superclass—**JPanel**. Select method **startAnimation** from the list and click the **Finish** button at the bottom of the **Connection Wizard** (Fig. 6.31). Repeat the above procedure for the **Stop Animation** button, but select method **stopAnimation** in Step 2 of the **Connection Wizard**.
Fig. 6.28 Connecting JButton and LogoAnimator.

Fig. 6.29 Connection Wizard dialog.
Fig. 6.30  Select **actionPerformed** event.

Fig. 6.31  Selecting method **startAnimation** for the target component.
To test that the connections between the buttons and the LogoAnimator work correctly, execute the `AnimationWindow` application by right clicking `AnimationWindow` in the Explorer window and selecting Execute from the menu (Fig. 6.32). Forte switches to the Running tab and displays `AnimationWindow` (Fig. 6.33). `AnimationWindow` contains the LogoAnimator and the two JButton. Click the Stop Animation button. The Deitel logo in LogoAnimator stops. Clicking the Start Animation button starts the animation from the point it stopped.

**Testing and Debugging Tip 6.1**

A benefit of working in a bean-ready development environment is that the beans typically execute live in the development environment. This allows you to view your program immediately in the design environment, rather than using the standard edit, compile and execute cycle.
6.3 Preparing a Class to be a JavaBean

In the previous section, we introduced the **LogoAnimator** JavaBean to demonstrate the basics of using JavaBeans within the Forte integrated development environment. This section presents the Java code for **LogoAnimator** (Fig. 6.34).

```java
// Fig. 6.34: LogoAnimator.java
// LogoAnimator is a JavaBean containing an animated logo.
package com.deitel.advjhtp1.beans;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

// Java extension packages
import javax.swing.*;

public class LogoAnimator extends JPanel
    implements ActionListener, Serializable {

    protected ImageIcon images[];
    protected int totalImages = 30, currentImage;
    protected Timer animationTimer;

    // load images and start animation
    public LogoAnimator() {
        images = new ImageIcon[ totalImages ];
        URL url;

        // load animation frames
        for ( int i = 0; i < images.length; ++i ) {
            url = LogAnimator.class.getResource(
                "images/deitel" + i + ".png" );
            images[ i ] = new ImageIcon( url );
        }
        startAnimation();
    }

    // render one frame of the animation
    public void paintComponent( Graphics g ) {
        super.paintComponent( g );
        // draw current animation frame
        images[ currentImage ].paintIcon( this, g, 0, 0 );
        currentImage = ( currentImage + 1 ) % totalImages;
    }

Fig. 6.34  Definition of class LogoAnimator (part 1 of 3).
```
// start Timer that drives animation
public void startAnimation()
{
    // if animationTimer is null, restart animation
    if ( animationTimer == null ) {
        currentImage = 0;
        animationTimer = new Timer( 50, this );
        animationTimer.start();
    }

    else // continue from last image displayed
    if ( !animationTimer.isRunning() )
        animationTimer.restart();
}

// repaint when Timer event occurs
public void actionPerformed( ActionEvent actionEvent )
{
    repaint();
}

// stop Timer that drives animation
public void stopAnimation()
{
    animationTimer.stop();
}

// get animation preferred width and height
public Dimension getPreferredSize()
{
    return new Dimension( 160, 80 );
}

// get animation minimum width and height
public Dimension getMinimumSize()
{
    return getPreferredSize();
}

// execute bean as standalone application
public static void main( String args[] )
{
    // create new LogoAnimator
    LogoAnimator animation = new LogoAnimator();

    // create new JFrame with title "Animation test"
    JFrame application = new JFrame( "Animator test" );
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

Fig. 6.34 Definition of class LogoAnimator (part 2 of 3).
Class **LogoAnimator** (Fig. 6.34) implements the **LogoAnimator** JavaBean. **LogoAnimator** extends **JPanel** (line 14), making it a GUI component. Many JavaBeans are GUI components intended to be manipulated visually in a builder tool, such as Forte. In fact, most Java **Swing** components are JavaBeans, such as the **JButtons** we manipulated visually in Forte in the previous section. GUIs using **Swing** components can be developed quickly in Forte and other JavaBean-enabled builder tools.

Class **LogoAnimator** implements interface **Serializable** (line 15). The **Serializable** interface allows an instance of **LogoAnimator** to be saved as a file. By implementing **Serializable**, a customized JavaBean can be saved and reloaded in a builder tool or in a Java application. Forte can save an instance of **LogoAnimator** with the **Serialize As...** option in the **Customize Bean** dialog. **Serializable** objects can be serialized in Java programs with the **ObjectOutputStream** and **ObjectInputStream** classes.

Software Engineering Observation 6.2

JavaBeans must all implement interface **Serializable** to support persistence using standard Java serialization.

Line 17 declares **ImageIcon** array **images**, which contains the 30 PNG images that comprise the animated logo in **LogoAnimator**. Line 18 declares two integer variables, **totalImages** and **currentImage**. Line 19 declares **animationTimer**, a **Timer** object that controls the animation speed.

Lines 22–36 define **LogoAnimator**’s no-argument constructor. Line 24 initializes array **images** with length **totalImages**. Lines 29–33 load the PNG images into the array. Line 35 invokes method **startAnimation** to start the **LogoAnimator** animation.

**LogoAnimator** overrides method **paintComponent** (lines 39–46), inherited from class **JPanel**. Method **paintComponent** draws **LogoAnimator** whenever it is called. Line 44 calls method **paintIcon** on an **ImageIcon** in array **images**. The **ImageIcon** is at index **currentImage**. This paints one of the animation frames. Line 45 advances the variable **currentImage** to the next animation frame.

Method **startAnimation** (lines 49–62) initializes **animationTimer**, the **Timer** object that controls the delay between animation frames. If **animationTimer** is null, lines 52–56 create a new **Timer** object with a delay of 50 milliseconds. Otherwise, **startAnimation** restarts **animationTimer** on lines 58–61.

**LogoAnimator** implements method **actionPerformed** of interface **ActionListener** on lines 65–68. The **animationTimer** generates an **ActionEvent** at a
rate specified in its constructor argument. When **animationTimer** generates an **ActionEvent**, line 67 calls method **repaint**. Method **repaint**, in turn, calls **paintComponent**, which draws the next animation frame.

Method **stopAnimation** (lines 71–74) calls method **stop** on **animationTimer** (line 73). This stops **animationTimer** from generating **ActionEvents**, which stops the animation.

Method **getPreferredSize** (lines 77–80) returns a **Dimension** object with the preferred size of **LogoAnimator**. Method **getMinimumSize** (line 83–86) simply calls **getPreferredSize**. The **LayoutManager** calls these two methods to determine how to size **LogoAnimator** within the runtime environment.

Method **main** (lines 89–106) allows **LogoAnimator** to be executed as an application. Method **main** creates a new **JFrame** and adds an instance of **LogoAnimator** to the **JFrame**. JavaBeans do not need a **main** method, but the **main** method is needed to execute a JavaBean independently.

**LogoAnimator** can be compiled either from the command line or in Forte. **LogoAnimator** declares a package, so use the **-d** option of the Java compiler to create the proper directory structure. The command line

```
javac -d . LogoAnimator.java
```

compiles **LogoAnimator** and places the package directory in the current directory. The full directory structure for the package is `com\deitel\advjhtp1\beans\` (substitute forward slashes, `/`, in UNIX/Linux). For **LogoAnimator** to execute properly, the directory **images** with the PNG files used by **LogoAnimator** must be placed in the same directory as class **LogoAnimator** (e.g. `com\deitel\advjhtp1\beans\`). The **images** directory and the PNG files for **LogoAnimator** are on the CD-ROM that accompanies this book.

To compile in Forte, open the **LogoAnimator.java** file in Forte, right click in the **Source Editor** window and select **Compile** (Fig. 6.35). Forte compiles the source code and reports any errors in a separate window. Be sure to place the **images** directory in the same directory as **LogoAnimator.class**. **LogoAnimator** will not execute properly without the images.

### 6.4 Creating a JavaBean: Java Archive Files

JavaBeans normally are stored and distributed in a **Java Archive files (JAR files)**. A JAR file for a JavaBean must contain a **manifest file**, which describes the JAR file contents. Manifest files contain attributes (called **headers**) that describe the individual components in the JAR. This is important for integrated development environments that support JavaBeans. When a JAR file containing a JavaBean (or a set of JavaBeans) is loaded into an IDE, the IDE reads at the manifest file to determine which of the classes in the JAR represent JavaBeans. IDEs typically make these classes available to the programmer in a visual manner, as shown in the Forte overview earlier in this chapter. We create file **manifest.tmp**, which the **jar** utility uses to create the file as **MANIFEST.MF** and places in the **META-INF** directory of the JAR file. [**Note:** The file **manifest.tmp** can have any name—**jar** simply uses the file’s contents to create **MANIFEST.MF** in the JAR file.] All JavaBean-aware development environments know to read the **MANIFEST.MF**
file in the **META-INF** directory of the JAR file. The Java interpreter can execute an application directly from a JAR file if the manifest file specifies which class in the JAR contains method `main`. Figure 6.36 shows the manifest file (`manifest.tmp`) for the *LogoAnimator* JavaBean.

**Software Engineering Observation 6.3**

You must define a manifest file that describes the contents of a JAR file if you intend either to use the bean in a bean-aware integrated development environment or execute an application directly from a JAR file.

```
1 Main-Class: com.deitel.advjhtp1.beans.LogoAnimator
2 Name: com/deitel/advjhtp1/beans/LogoAnimator.class
3 Java-Bean: True
```

Fig. 6.36  Method file `manifest.tmp` for the *LogoAnimator* bean.
Class `com.deitel.advjhtp1.beans.LogoAnimator` contains method `main`. This is specified by the `Main-Class` header (line 1). This header enables the virtual machine to execute the application in the JAR file directly. To execute `LogoAnimator` from its JAR file, launch the Java virtual machine with the `-jar` command-line option as follows:

```java
java -jar LogoAnimator.jar
```

The interpreter looks at the manifest file to determine which class to execute. On many platforms, you can execute an application in a JAR file by double clicking the JAR file in your system’s file manager. This executes the `jar` command with the `-jar` option for the JAR file the user clicks. The application can also be executed from a JAR file that does not contain a manifest with the command

```java
java -classpath LogoAnimator.jar
```

where `-classpath` indicates the class path (i.e., the directories and JAR files in which the interpreter should search for classes). The `-classpath` option is followed by the JAR file containing the application class. The last command-line argument is the full class name (including the package name) for the application class.

Line 3 of the manifest file specifies the `Name` header of the file containing the bean class (including the `.class` file name extension), using its package and class name. Notice that the dots (.) typically used in package names are replaced with forward slashes (/) for the `Name` header in the manifest file. Line 4 use the `Java-Bean` header to specify that the class named on line 3 is a JavaBean. It is possible to have classes that are not JavaBeans in a JAR file. Such classes typically support the JavaBeans in the archive. For example, a linked-list bean might have a supporting linked-list-node class, objects of which represent each node in the list. Each class listed in the manifest file should be separated from other classes by a blank line. If the class is a bean, its `Name` header should be followed immediately by its `Java-Bean` header.

In the manifest file, a bean’s name is specified with the `Name` header followed by the fully qualified name of the bean (i.e., the complete package name and class name). The dots (.) normally used to separate package names and class names are replaced with forward slash (/) in this line of the manifest file.

---

**Common Programming Error 6.1**

If a class represents a bean, the `Java-Bean` header must follow the `Name` header immediately with a value of `True`. Otherwise, IDEs will not recognize the class as a bean.

---

**Software Engineering Observation 6.4**

If a class containing `main` is included in a JAR file, that class can be used by the interpreter to execute the application directly from the JAR file by specifying the `Main-Class` header at the beginning of the manifest file. The full package name and class name of the class should be specified with periods (.) separating the package components and class name.

---

**Common Programming Error 6.2**

Not specifying a manifest file or specifying a manifest file with incorrect syntax when creating a JAR file is an error—builder tools will not recognize the beans in the JAR file.
If a JAR file manifest does not specify the Main-Class header, there must be a blank line at the top of the manifest file before listing any Name headers. Some JAR utilities will report an error and not create a JAR without a blank line at the top of the manifest.

Next, we create the JAR file for the LogoAnimator bean. This is accomplished with the jar utility at the command line (such as the MS-DOS prompt or UNIX shell). The command

```bash
jar cfm LogoAnimator.jar manifest.tmp
com\deitel\advjhtp1\beans\*.*
```

creates the JAR file. [Note: This command uses the backslash (\) as the directory separator from the Windows Command Prompt. UNIX would use the forward slash (/) as the directory separator.] In the preceding command, jar is the Java archive utility used to create JAR files. The options for the jar utility—cfm are provided next. The letter c indicates that we are creating a JAR file. The letter f indicates that the next argument in the command line (LogoAnimator.jar) is the name of the JAR file to create. The letter m indicates that the next argument in the command line (manifest.tmp) is the manifest file that jar uses to create the file META-INF/MANIFEST.MF in the JAR. Following the options, the JAR file name and the manifest file name are the actual files to include in the JAR file. We specified com\deitel\advjhtp1\beans\*.*, indicating that all the files in the beans directory should be included in the JAR file. The com.deitel.advjhtp1.beans package directory contains the .class files for the LogoAnimator and its supporting classes, as well as the images used in the animation. [Note: You can include particular files by specifying the path and file name for each individual file.] It is important that the directory structure in the JAR file match the class’ package structure. Therefore, we executed the jar command from the directory on our system in which the com directory that begins the package name reside.

To confirm that the files were archived correctly, issue the command

```bash
jar tvf LogoAnimator.jar
```

In this command, the letter t indicates that jar should list the table of contents for the JAR file. The letter v indicates that the output should be verbose (the verbose output includes the file size in bytes and the date and time each file was created, in addition to the directory structure and file name). The letter f specifies that the next argument on the command line is the JAR file for which jar should display information.

Try executing the LogoAnimator application with the command

```bash
java -jar LogoAnimator.jar
```

You will see that the animation appears in its own window on your screen.

JAR files also can be created inside Forte’s integrated development environment. Right click LogoAnimator in the Explorer window and select Add to JAR from the Tools menu, as described in Section 6.2. This displays the JAR Packager dialog. Our class LogoAnimator already is selected to be included in the JAR file. At the top of the dialog, type LogoAnimator.jar in the JAR Archive textfield and specify the directory in which the JAR will appear. Add the images directory to the JAR (Fig. 6.37). Next, click the Manifest tab and select the Generate File List check box. This creates a list of
all files in the JAR, including `LogoAnimator.class` and the PNG files. The `Java-Bean` and `Main-Class` headers are not generated by Forte and must be typed into the manifest (Fig. 6.38). Click `Create JAR` to create `LogoAnimator.jar`. Now the Logo-Animator can be loaded into the `Component Palette` as described in Section 6.2.

---

**Fig. 6.37** Add `images` directory to `LogoAnimator.jar`.

**Fig. 6.38** Manifest tab of JAR Packager dialog.
6.5 JavaBean Properties

In this section, we demonstrate adding an animationDelay property to LogoAnimator, to control the animation’s speed. For this purpose, we extend class LogoAnimator to create class LogoAnimator2. The new code for our property is defined by methods setAnimationDelay (lines 16–19) and getAnimationDelay (lines 22–25) in Fig. 6.39.

```java
// Fig. 6.39: LogoAnimator2.java
// LogoAnimator2 extends LogoAnimator to include
// animationDelay property and implements ColorListener
package com.deitel.advjhtp1.beans;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class LogoAnimator2 extends LogoAnimator {

    // set animationDelay property
    public void setAnimationDelay( int delay )
    {
        animationTimer.setDelay( delay );
    }

    // get animationDelay property
    public int getAnimationDelay()
    {
        return animationTimer.getDelay();
    }

    // launch LogoAnimator in JFrame for testing
    public static void main( String args[] )
    {
        // create new LogoAnimator2
        LogoAnimator2 animation = new LogoAnimator2();

        // create new JFrame and add LogoAnimator2 to it
        JFrame application = new JFrame( "Animator test" );
        application.getContentPane().add( animation,
                                           BorderLayout.CENTER );
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        application.pack();
        application.setVisible( true );
    }
}
```

Fig. 6.39 LogoAnimator2 with property animationDelay.
To create the `animationDelay` property, we defined methods `setAnimationDelay` and `getAnimationDelay`. A read/write property of a bean is defined as a `set/get` method pair of the form

```java
public void setPropertyName( DataType value )
public DataType getPropertyName()
```

where `PropertyName` is replaced in each case by the actual property name. These methods often are referred to as a “property set method” and “property get method,” respectively.

### Software Engineering Observation 6.5

A JavaBean read/write property is defined by a `set/get` method pair in which the `set` method returns `void` and takes one argument and the `get` method returns the same type as the corresponding `set` method’s argument and takes no arguments. It is also possible to have read-only properties (defined with only a `get` method) and write-only properties (defined with only a `set` method).

### Software Engineering Observation 6.6

For a property with the name `propertyName`, the corresponding `set/get` method pair would be `setPropertyName/getPropertyName` by default. Note that the first letter of `propertyName` is capitalized in the `set/get` method names.

If the property is a `boolean` data type, the `set/get` method pair is sometimes defined as

```java
public void setPropertyName( boolean value )
public boolean isPropertyName()
```

where the `get` method name begins with the word `is` rather than `get`.

When a builder tool examines a bean, it inspects the bean methods for pairs of `set/get` methods that represent properties (some builder tools also expose read-only and write-only properties). This process is known as introspection. If the builder tool finds an appropriate `set/get` method pair during the introspection process, the builder tool exposes that pair of methods as a property in the builder tool’s user interface. In the first `LogoAnimator`, the pair of methods

```java
public void setBackground( Color c )
public Color getBackground()
```

that were inherited from class `JPanel` allowed Forte to expose the `background` property in the `Component Inspector` for customization. Notice that the naming convention for the `set/get` method pair used a capital letter for the first letter of the property name, but the exposed property in the `Component Inspector` is shown with a lowercase first letter.

### Software Engineering Observation 6.7

When a builder tool examines a bean, if it locates a `set/get` method pair that matches the JavaBean’s property pattern, it exposes that pair of methods as a property in the bean.

Remember that class `LogoAnimator2` must be wrapped as a JavaBean to load it into Forte and other builder tools. Compile `LogoAnimator2`, and then place it in a JAR file as described in the previous section. Now import `LogoAnimator2` into the `Component Palette`, and drop an instance of `LogoAnimator2` into a `JFrame` as in Section 6.2. Select `LogoAnimator2` in the `Form` window or `Component Inspector`. The `animationDelay` property is now exposed in the `Component Inspector` (Fig. 6.40). Try
changing the value of the property to see its effect on the speed of the animation (you must press Enter after changing the value to effect the change). Smaller values cause the animation to spin faster, larger values cause it to spin slower. Try typing 1000 to see one frame of the animation per second.

### 6.6 Bound Properties

A bound property causes the JavaBean that owns the property to notify other objects when the bound property’s value changes. This notification is accomplished with standard Java event-handling features—the bean notifies its registered PropertyChangeListeners when the bound property’s value changes. To support this feature, the java.beans package provides interface PropertyChangeListener so listeners can be configured to receive property-change notifications, class PropertyChangeEvent to provide information to a PropertyChangeListener about the change in a property’s value and class PropertyChangeSupport to provide the listener registration and notification services (i.e., to maintain the list of listeners and notify them when an event occurs).

**Software Engineering Observation 6.8**

A bound property causes the object that owns the property to notify other objects that there has been a change in the value of that property.

The next example creates a new GUI component (SliderFieldPanel) that extends JPanel and includes one JSlider object and one JTextField object. When the JSlider value changes, our new GUI component automatically updates the JTextField with the new value. Also, when a new value is entered in the JTextField and the user presses the Enter key, the JSlider is automatically repositioned to the appro-
appropriate location. Our purpose in defining this new component is to link one of these to the LogoAnimator2 animation to control the speed of the animation. When the SliderFieldPanel value changes, we want the animation speed to change. Figure 6.41 presents the code for class SliderFieldPanel.

```java
package com.deitel.advjhtp1.beans;

// Fig. 6.41: SliderFieldPanel.java
// SliderFieldPanel provides a slider to adjust the animation speed of LogoAnimator2.

public class SliderFieldPanel extends JPanel
    implements Serializable {

    private JSlider slider;
    private JTextField field;
    private Box boxContainer;
    private int currentValue;

    // object to support bound property changes
    private PropertyChangeSupport changeSupport;

    // SliderFieldPanel constructor
    public SliderFieldPanel() {
        // create PropertyChangeSupport for bound properties
        changeSupport = new PropertyChangeSupport( this );

        // initialize slider and text field
        slider = new JSlider( SwingConstants.HORIZONTAL, 1, 100, 1 );
        field = new JTextField( String.valueOf( slider.getValue() ), 5 );

        // set box layout and add slider and text field
        boxContainer = new Box(BoxLayout.X_AXIS);
        boxContainer.add( slider );
        boxContainer.add( Box.createHorizontalStrut( 5 ) );
        boxContainer.add( field );
        boxContainer.setLayout( new BorderLayout() );
        add( boxContainer );
    }
}
```

Fig. 6.41 Definition for class SliderFieldPanel (part 1 of 4).
// add ChangeListener for JSlider
slider.addChangeListener(
    new ChangeListener() {
        // handle state change for JSlider
        public void stateChanged(ChangeEvent changeEvent) {
            setCurrentValue(slider.getValue());
        }
    } // end anonymous inner class
); // end call to addChangeListener

// add ActionListener for JTextField
field.addActionListener(
    new ActionListener() {
        // handle action for JTextField
        public void actionPerformed(ActionEvent actionEvent) {
            setCurrentValue(Integer.parseInt(field.getText()));
        }
    } // end anonymous inner class
); // end call to addActionListener

} // end SliderFieldPanel constructor

// add PropertyChangeListener
public void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
}

// remove PropertyChangeListener
public void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
}

// set minimumValue property
public void setMinimumValue(int minimum) {
    slider.setMinimum(minimum);
}
if ( slider.getValue() < slider.getMinimum() ) {
    slider.setValue( slider.getMinimum() );
    field.setText( String.valueOf( slider.getValue() ) );
}

get minimumValue property
public int getMinimumValue()
{
    return slider.getMinimum();
}

get maximumValue property
public void setMaximumValue( int maximum )
{
    slider.setMaximum( maximum );
    if ( slider.getValue() > slider.getMaximum() ) {
        slider.setValue( slider.getMaximum() );
        field.setText( String.valueOf( slider.getValue() ) );
    }
}

get maximumValue property
public int getMaximumValue()
{
    return slider.getMaximum();
}

set currentValue property
public void setCurrentValue( int current )
throws IllegalArgumentException
{
    if ( current < 0 )
        throw new IllegalArgumentException();
    int oldValue = currentValue;
    currentValue = current;
    slider.setValue( currentValue );
    field.setText( String.valueOf( currentValue ) );
    changeSupport.firePropertyChange( "currentValue", new Integer( oldValue ),
    new Integer( currentValue ) );
}

Definition for class SliderFieldPanel (part 3 of 4).
Class `SliderFieldPanel` (Fig. 6.41) begins by specifying that it will be part of the `com.deitel.advjhtp1.beans` package (line 4). The class is a subclass of `JPanel`, so we can add a `JSlider` and a `JTextField` to it. Objects of class `SliderFieldPanel` can then be added to other containers.

Lines 19–25 declare instance variables of type `JSlider` (`slider`) and `JTextField` (`field`) that represent the subcomponents the user will use to set the `SliderFieldPanel` value, a `Box` (`boxContainer`) that will manage the layout, an `int` (`currentValue`) that stores the current value of the `SliderFieldPanel` and a `PropertyChangeSupport` (`changeSupport`) that will provide the listener registration and notification services.

Line 31 creates the `PropertyChangeSupport` object. The argument `this` specifies that an object of this class (`SliderFieldPanel`) is the source of the `PropertyChangeEvent`. Lines 49–61 of the constructor register the `ChangeListener` for `slider`. When `slider`’s value changes, line 56 calls `setCurrentValue` to update `field` and notify registered `PropertyChangeListener`s of the change in

```java
152     // get currentValue property
153     public int getCurrentValue()
154     {
155         return slider.getValue();
156     }
157
158     // set fieldWidth property
159     public void setFieldWidth( int columns )
160     {
161         field.setColumns( columns );
162         boxContainer.validate();
163     }
164
165     // get fieldWidth property
166     public int getFieldWidth()
167     {
168         return field.getColumns();
169     }
170
171     // get minimum panel size
172     public Dimension getMinimumSize()
173     {
174         return boxContainer.getMinimumSize();
175     }
176
177     // get preferred panel size
178     public Dimension getPreferredSize()
179     {
180         return boxContainer.getPreferredSize();
181     }
182
183 } // end class SliderFieldPanel
```

Fig. 6.41 Definition for class `SliderFieldPanel` (part 4 of 4).
value. Similarly, lines 64–78 register the `ActionListener` for `field`. When `field`’s value changes, lines 72–73 call `setCurrentValue` to update `slider` and notify registered `PropertyChangeListener`s of the change in value.

To support registration of listeners for changes to our `SliderFieldPanel`’s bound property, we define methods `addPropertyChangeListener` (lines 83–87) and `removePropertyChangeListener` (lines 90–94). Each of these methods calls the corresponding method in the `PropertyChangeSupport` object `changeSupport`. This object provides the event notification services when the property value changes.

**Software Engineering Observation 6.9**

To define an event for a bean, you must supply a listener interface and an event class, and the bean must define methods that allow adding and removing of listeners. For bound property events, the listener interface and the event class are already defined (`PropertyChangeListener` and `PropertyChangeEvent`, respectively). A bean that supports bound-property events must define method `addPropertyChangeListener` and method `removePropertyChangeListener` to provide listener registration services.

Class `SliderFieldPanel` provides several properties. Methods `setMinimumValue` (lines 97–105) and `getMinimumValue` (lines 108–111) define property `minimumValue`. Property `maximumValue` is defined by methods `setMaximumValue` (lines 114–122) and `getMaximumValue` (lines 125–128). Methods `setFieldWidth` (lines 159–163) and `getFieldWidth` (lines 166–169) define property `fieldWidth`. Methods `getMinimumSize` (lines 172–175) and `getPreferredSize` (line 178–181) return the minimum size and preferred size of the `Box` object `boxContainer`, which manages the layout of the `JSlider` and `JTextField`.

**Look-and-Feel Observation 6.1**

If a bean will appear as part of a user interface, the bean should define method `getPreferredSize`, which takes no arguments and returns a `Dimension` object containing the preferred width and height of the bean. This helps the layout manager size the bean.

Methods `setCurrentValue` (lines 131–150) and `getCurrentValue` (lines 153–156) define the bound property `currentValue`. When the bound property changes, the registered `PropertyChangeListener`s must be notified of the change. The JavaBeans specification ([java.sun.com/products/javabeans/docs/spec.html](http://java.sun.com/products/javabeans/docs/spec.html)) requires that each bound-property listener be presented with the old and new property values when notified of the change (the values can be `null` if they are not needed). For this reason, line 137 saves the previous property value. Line 140 sets the new property value. Lines 143–144 ensure that the `JSlider` and `JTextField` show the appropriate new values. Lines 147–149 invoke the `PropertyChangeSupport` object’s `firePropertyChange` method to notify each registered `PropertyChangeListener`. The first argument is a `String` containing the property name that changed—`currentValue`. The second argument is the old property value. The third argument is the new property value.

**Software Engineering Observation 6.10**

`PropertyChangeListener`s are notified of a property-change event with both the old and the new value of the property. If these values are not needed, they can be `null`. 
Software Engineering Observation 6.11
Class PropertyChangeSupport is provided as a convenience to implement the listener registration and notification support for property-change events.

Remember that you should package the SliderFieldPanel class as a JavaBean to load it into a builder tool. Archive the class in a JAR file. The manifest file for this example is shown in Fig. 6.42. Line 2 specifies the name of the class file (com\deitel\advjhtp1\beans\SliderFieldPanel.class) that represents the bean. Line 3 specifies that the class named in line 1 is a JavaBean. There is no Main-Class header in this file, because the SliderFieldPanel is not an application. Finally, install SliderFieldPanel.jar into the builder tool.

To demonstrate the functionality of the bound property, place a SliderFieldPanel bean and a LogoAnimator2 bean into a JFrame. Select the SliderFieldPanel bean (Fig. 6.43), set its maximumValue property to 1000 and set its currentValue to 50 (the default animation speed for the LogoAnimator2). In Forte, select Connection Mode from the Component Palette. Click the SliderFieldPanel; then click LogoAnimator2. Red squares appear at the corners of each component and the Connection Wizard opens.

In Step 1 of the Connection Wizard, select propertyChange as the event for the source component and click the Next button (Fig. 6.44). Select LogoAnimator2’s animationDelay property in Step 2 of the Connection Wizard (Fig. 6.45) and click Next. Finally, select SliderFieldPanel’s currentValue property in Step 3 (Fig. 6.46) and click Finish. The animationDelay property is now bound to the SliderFieldPanel’s currentValue property. Execute the JFrame to see the connected LogoAnimator2 and SliderFieldPanel (Fig. 6.47).

Try adjusting the slider to see the animation speed change. Move the slider left to see the speed of the animation increase; move the slider right to see the animation speed decrease.

```plaintext
1
2 Name: com/deitel/advjhtp1/beans/SliderFieldPanel.class
3 Java-Bean: True
```

Fig. 6.42 Manifest file for the SliderFieldPanel JavaBean.

Fig. 6.43 Change properties currentValue and maximumValue.
Fig. 6.44  Select \texttt{propertyChange} event.

Fig. 6.45  Select \texttt{animationDelay} property of \texttt{LogoAnimator2}.

Fig. 6.46  Select \texttt{currentValue} Bound Property.
6.7 Indexed Properties and Custom Events

Although standard properties, bound properties and standard Java events provide a great deal of functionality, JavaBeans can be further customized with other types of properties and programmer-defined events. An indexed property is like a standard property, except that the indexed property is an array of primitives or objects. Two get and two set methods define an indexed property. The get methods are of the form

\[
\text{public Datatype[]} \text{ setPropertyName()} \\
\text{public Datatype setPropertyName( int index )}
\]

The first get method returns the entire array of an indexed property. The second get method returns the item at the array index indicated by the get method’s parameter. The set methods are of the form

\[
\text{public void setPropertyName( Datatype[]} \text{ data)} \\
\text{public void setPropertyName( int index, Datatype data )}
\]

The first set method sets the indexed property to the value of the argument. The second set method sets the item at the indicated array index to the value of the second parameter.

Software Engineering Observation 6.12

An indexed property functions like a regular property and is exposed in the property sheet like normal properties.

A JavaBean can generate programmer-defined events. A programmer-defined event, or custom event, provides functionality that standard Java events do not provide. An event class extends \textit{java.util.EventObject} and the listener interface extends \textit{java.util.EventListener}.

Our next example demonstrates an indexed property and a custom event. In the example we create a \textit{ColorSliderPanel} that enables a user to choose values for the red, green and blue parts of a color. The \textit{ColorSliderPanel} maintains these three integer values in an indexed property and uses them to create \textit{Color} objects. This custom GUI component also generates custom \textit{ColorEvents}, so that it can notify its registered listeners when the user changes the color.

We begin by defining an \textit{EventObject} class and an \textit{EventListener} interface for the custom event \textit{ColorEvent}. Figure 6.48 shows the \textit{ColorEvent} class and Fig. 6.49 shows the \textit{ColorListener} interface. Class \textit{ColorEvent} is a custom event that extends class \textit{EventObject}. Parameter \textit{color} in the constructor (lines 15–19) represents the value of the \textit{ColorEvent}'s \textit{color} property. Method \textit{setColor} (lines 22–25) sets the \textit{color} instance variable. Method \textit{getColor} (lines 28–31) returns the \textit{color} property.
Interface **ColorListener** (Fig. 6.49) is a custom listener interface that extends class **EventListener**. Classes that implement **ColorListener** listen for **ColorEvents**. The **ColorEvent** event source calls its registered listeners’ **colorChanged** method (declared at line 11) with a **ColorEvent** object describing the change. All listeners for **ColorEvents** must implement the **ColorListener** interface.
Class ColorSliderPanel (Fig. 6.50) is a JavaBean that issues colorChanged ColorEvents when the sliders change the color value. ColorSliderPanel consists of three SliderFieldPanels marked with JLabels as Red, Green and Blue. Three JTextFields, one in each SliderFieldPanel, display integers from zero through 255. The three values are stored in an indexed property called redGreenBlue. Moving a JSlider changes the number displayed in its JTextField and changes the value of redGreenBlue. Changing redGreenBlue’s value causes ColorSliderPanel to fire a ColorEvent. The ColorEvent contains a Color object initialized to the three values of redGreenBlue. Our purpose in defining ColorSliderPanel is to link one of these to the LogoAnimator2 animation, to change the background color of LogoAnimator2.
// constructor for ColorSliderPanel
public ColorSliderPanel()
{
    // initialize redGreenBlue property
    redGreenBlue = new int[] { 0, 0, 0 };

    // initialize gui components for red slider
    redLabel = new JLabel( "Red:" );
    redSlider = new SliderFieldPanel();
    redSlider.setMinimumValue( 0 );
    redSlider.setMaximumValue( 255 );

    // initialize gui components for green slider
    greenLabel = new JLabel( "Green: " );
    greenSlider = new SliderFieldPanel();
    greenSlider.setMinimumValue( 0 );
    greenSlider.setMaximumValue( 255 );

    // initialize gui components for blue slider
    blueLabel = new JLabel( "Blue:" );
    blueSlider = new SliderFieldPanel();
    blueSlider.setMinimumValue( 0 );
    blueSlider.setMaximumValue( 255 );

    // set layout and add components
    setLayout( new BorderLayout() );

    labelPanel = new JPanel( new GridLayout( 3, 1 ) );
    labelPanel.add( redLabel );
    labelPanel.add( greenLabel );
    labelPanel.add( blueLabel );

    sliderPanel = new JPanel( new GridLayout( 3, 1 ) );
    sliderPanel.add( redSlider );
    sliderPanel.add( greenSlider );
    sliderPanel.add( blueSlider );

    add( labelPanel, BorderLayout.WEST );
    add( sliderPanel, BorderLayout.CENTER );

    // add PropertyChangeListener for redSlider
    redSlider.addPropertyChangeListener( new PropertyChangeListener() {
    // handle propertyChange for redSlider
        public void propertyChange( PropertyChangeEvent propertyChangeEvent )
        {
            setRedGreenBlue( RED_INDEX, redSlider.getCurrentValue() );
        }
    } );

Fig. 6.50  Definition of class ColorSliderPanel (part 2 of 5).
} // end anonymous inner class
38
}; // end call to addPropertyChangeListener
39
// add PropertyChangeListener for greenSlider
40
greenSlider.addPropertyChangeListener(
41
new PropertyChangeListener() {
42
    // handle propertyChange for greenSlider
43
    public void propertyChange( PropertyChangeEvent
44         propertyChangeEvent )
45    {
46        setRedGreenBlue( GREEN_INDEX,
47            greenSlider.getCurrentValue() );
48    }
49
} // end anonymous inner class
50
}; // end call to addPropertyChangeListener
51
// add PropertyChangeListener for blueSlider
52
blueSlider.addPropertyChangeListener(
53
new PropertyChangeListener() {
54
    // handle propertyChange for blueSlider
55
    public void propertyChange( PropertyChangeEvent
56         propertyChangeEvent )
57    {
58        setRedGreenBlue( BLUE_INDEX,
59            blueSlider.getCurrentValue() );
60    }
61
} // end anonymous inner class
62
}; // end call to addPropertyChangeListener
63
} // end ColorSliderPanel constructor
64
// add ColorListener
65
public void addColorListener(
66    ColorListener colorListener )
67    {
68        // listeners must be accessed atomically
69        synchronized ( listeners ) {
70            listeners.add( colorListener );
71        }
72    }
// remove ColorListener
public void removeColorListener(ColorListener colorListener) {
    // listeners must be accessed by one thread only
    synchronized (listeners) {
        listeners.remove(colorListener);
    }
}

// fire ColorEvent
public void fireColorChanged() {
    Iterator iterator;

    // listeners must be accessed atomically
    synchronized (listeners) {
        iterator = new HashSet(listeners).iterator();
    }

    // create new Color with values of redGreenBlue
    // create new ColorEvent with color variable
    Color color = new Color(redGreenBlue[RED_INDEX],
        redGreenBlue[GREEN_INDEX],
        redGreenBlue[BLUE_INDEX]);
    ColorEvent colorEvent = new ColorEvent(this, color);

    // notify all registered ColorListeners of ColorChange
    while (iterator.hasNext()) {
        ColorListener colorListener = (ColorListener)
            iterator.next();
        colorListener.colorChanged(colorEvent);
    }
}

// get redGreenBlue property
public int[] getRedGreenBlue() {
    return redGreenBlue;
}

// get redGreenBlue indexed property
public int getRedGreenBlue(int index) {
    return redGreenBlue[index];
}

// set redGreenBlue property
public void setRedGreenBlue(int[] array) {
    redGreenBlue = array;
}

Fig. 6.50 Definition of class ColorSliderPanel (part 4 of 5).
Lines 31–121 contain the constructor for ColorSliderPanel. Line 34 initializes the redGreenBlue indexed property. Lines 37–52 initialize the JLabels’ components and SliderFieldPanels. Each part of property redGreenBlue has a JLabel and a SliderFieldPanel associated with it. The SliderFieldPanels’ JSliders are set to a range of 0 through 255, and the JTextFields are set with the initial value of the JSliders. Line 55 sets the layout to BorderLayout. Lines 57–60 add the JLabels to a new JPanel with a three-by-one GridLayout. Lines 62–65 add the SliderFieldPanels to a new JPanel with a three-by-one GridLayout. Lines 67–68 add the JPanels to ColorSliderPanel. Lines 71–119 add PropertyChangeListener to the SliderFieldPanels. Each call to addPropertyChangeListener creates an instance of a PropertyChangeListener anonymous inner class. When a SliderFieldPanel fires a PropertyChangeEvent, the propertyChanged method of the appropriate PropertyChangeListener updates the value of indexed property redGreenBlue.

Methods addColorListener (lines 124–131) and removeColorListener (lines 134–141) contain synchronized blocks in which the Set listeners (line 28) is modified. Set listeners contains all the registered listeners of type ColorListener. Method fireColorChanged (lines 144–166) uses method iterator to create an Iterator from listeners. Lines 155–158 create a ColorEvent object with a Color attribute matching the values of the redGreenBlue property. Method fireColorChanged then sends the event to all registered listeners by calling method colorChanged on every listener.

Lines 169–191 contain methods to manipulate the redGreenBlue property. Method getRedGreenBlue (lines 169–172) with no parameters returns the integer array redGreenBlue. Method getRedGreenBlue (lines 175–178) with an integer parameter returns the value of redGreenBlue at the index of the parameter. Property redGreenBlue can be set with two versions of method setRedGreenBlue. Method setRedGreenBlue (lines 181–184) with an integer array parameter sets redGreenBlue to the parameter. Method setRedGreenBlue (lines 187–191) with integer parameters index and value sets the value of redGreenBlue at index. This version of the method also calls fireColorChanged to generate a ColorEvent.

Classes ColorEvent, ColorListener and ColorSliderPanel should be packaged in a JAR file so ColorSliderPanel can be used as a JavaBean. Figure 6.51 shows the manifest file for this example. Line 2 specifies the name of the class file

```java
186     // set redGreenBlue indexed property
187     public void setRedGreenBlue(int index, int value)
188     {
189         redGreenBlue[index] = value;
190         fireColorChanged();
191     }
192
193     }  // end class ColorSliderPanel
```

Fig. 6.50 Definition of class ColorSliderPanel (part 5 of 5).
(com\deitel\advjhtp1\beans\ColorSliderPanel.class) that represents the bean. Line 3 specifies that the class named in line 2 is a JavaBean. There is no Main-Class header line in this file, because ColorSliderPanel is not an application. No entries are listed for ColorEvent and ColorListener, because they are only supporting classes.

Install ColorSliderPanel into the Component Palette and drop instances of LogoAnimator2 and ColorSliderPanel into a JFrame. Switch to Connection Mode and click ColorSliderPanel then LogoAnimator2. The Connection Wizard opens with a list of events from which to choose. Select colorChanged from the menu and click the Next button (Fig. 6.52). In Step 2, click the Method radio button and select method setBackground (Fig. 6.53). This method of LogoAnimator2 will be called with the ColorEvent’s Color property as the argument. In Step 3, click the User Code: radio button and type evt.getColor() into the text area then click Finish (Fig. 6.54). This line calls ColorEvent’s getColor method, which returns a Color object. LogoAnimator2 now listens for ColorEvents generated by ColorSliderPanel.

Execute the JFrame to see LogoAnimator2 and ColorSliderPanel. Try adjusting the three different sliders. Each slider changes one of the elements of the Color object of the background of LogoAnimator2. Try moving the sliders to change the background color and try entering a new value as text. Figure 6.55 shows several of the possible colors.

| 1 | Name: com/deitel/advjhtp1/beans/ColorSliderPanel.class |
| 2 | Java-Bean: True |

Fig. 6.51 Manifest file for the ColorSliderPanel JavaBean.

---

Fig. 6.52 Selecting colorChanged method in Connection Wizard.
**Fig. 6.53** Selecting `setBackground` method for target `LogoAnimator2`.

**Fig. 6.54** Entering user code in Connection Wizard.

**Fig. 6.55** Using the `ColorSliderPanel` to change the `background` color of `LogoAnimator2`. 
6.8 Customizing JavaBeans for Builder Tools

As mentioned previously, builder tools use Java’s introspection mechanism to expose a JavaBean’s properties, methods and events if the programmer follows the proper JavaBean design patterns (such as the special naming conventions discussed for set/get method pairs that define bean properties). Builder tools use the classes and interfaces of package java.lang.reflect to perform introspection. For JavaBeans that do not follow the JavaBean design patterns, or for JavaBeans in which the programmer wants to customize the exposed set of properties, methods and events, the programmer can supply a class that implements interface BeanInfo (package java.beans). The BeanInfo class describes to the builder tool how to present the features of the bean to the programmer.

Software Engineering Observation 6.13

A JavaBean’s properties, methods and events can be exposed by a builder tool if the programmer follows the proper JavaBean design patterns.

Software Engineering Observation 6.14

Every BeanInfo class must implement interface BeanInfo. This interface describes the methods used by a builder tool to determine the features of a bean.

Class SliderFieldPanel (Fig. 6.41) exposes many properties and events when it is selected in the Component Inspector or connected with the Connection Wizard. For this bean, we want the programmer to see only properties fieldWidth, currentValue, minimumValue and maximumValue (the other properties were inherited from class JPanel and are not truly relevant to our bean). Also, the only event we want the programmer to use for our component is the bound-property event.

Software Engineering Observation 6.15

By convention, the BeanInfo class has the same name as the bean and ends with BeanInfo and is placed in the same package as the bean it describes or else it will not be found automatically.

Software Engineering Observation 6.16

By convention, the BeanInfo class is included in the same JAR as the SliderFieldPanel JavaBean. When the bean is loaded, the builder tool determines whether the JAR file contains a BeanInfo class for a bean. If a BeanInfo class is found, it is used to determine the exposed features of the bean. Otherwise, standard introspection is used to determine the exposed features of the bean.

Figure 6.56 presents class SliderFieldPanelBeanInfo to customize the properties and events exposed in builder tools for our SliderFieldPanel bean. The screen captures in Fig. 6.57 show the exposed features of the SliderFieldPanel JavaBean.

```java
// Fig. 6.56 SliderFieldPanelBeanInfo.java
// SliderFieldPanelBeanInfo is the BeanInfo class for
// SliderFieldPanel
package com.deitel.advjhtp1.beans;
```

Fig. 6.56 SliderFieldPanelBeanInfo exposes properties and events for SliderFieldPanel (part 1 of 5).
// Java core packages
import java.beans.*;
import java.awt.Image;

// Java extension packages
import javax.swing.*;

public class SliderFieldPanelBeanInfo extends SimpleBeanInfo {

    public static final Class beanClass = SliderFieldPanel.class;

    // return general description of bean
    public BeanDescriptor getBeanDescriptor() {
        BeanDescriptor descriptor = new BeanDescriptor(
                    beanClass, SliderFieldPanelCustomizer.class);
        descriptor.setDisplayName( "Slider Field" );
        descriptor.setShortDescription(
                "A slider bar to change a numerical property." );

        return descriptor;
    }

    // return bean icon
    public Image getIcon( int iconKind ) {
        Image image = null;

        switch( iconKind ) {
            case ICON_COLOR_16x16:
                image = loadImage( "icon1.gif" );
                break;
            case ICON_COLOR_32x32:
                image = loadImage( "icon2.gif" );
                break;
            case ICON_MONO_16x16:
                image = loadImage( "icon3.gif" );
                break;
            case ICON_MONO_32x32:
                image = loadImage( "icon4.gif" );
                break;
            default:
                break;
        }
        return image;
    }
}

Fig. 6.56  SliderFieldPanelBeanInfo exposes properties and events for
SliderFieldPanel (part 2 of 5).
return image;

// return array of MethodDescriptors for public get methods
// of class SliderFieldPanel
public MethodDescriptor[] getMethodDescriptors()
{
    // create array of MethodDescriptors
    try {
        MethodDescriptor getMinimumValue = new MethodDescriptor( beanClass.getMethod( "getMinimumValue", null ) );
        MethodDescriptor getMaximumValue = new MethodDescriptor( beanClass.getMethod( "getMaximumValue", null ) );
        MethodDescriptor getCurrentValue = new MethodDescriptor( beanClass.getMethod( "getCurrentValue", null ) );
        MethodDescriptor getFieldWidth = new MethodDescriptor( beanClass.getMethod( "getFieldWidth", null ) );
        MethodDescriptor[] descriptors = { getMinimumValue, getMaximumValue, getCurrentValue, getFieldWidth };

        return descriptors;
    }

    // printStackTrace if NoSuchMethodException thrown
    catch ( NoSuchMethodException methodException ) {
        methodException.printStackTrace();
    }

    // printStackTrace if SecurityException thrown
    catch ( SecurityException securityException ) {
        securityException.printStackTrace();
    }

    return null;
}

// return PropertyDescriptor array
public PropertyDescriptor[] getPropertyDescriptors() throws RuntimeException
{
    // create array of PropertyDescriptors
    try {
        // printStackTrac...
// fieldWidth property
PropertyDescriptor fieldWidth = new PropertyDescriptor( "fieldWidth", beanClass );
fieldWidth.setShortDescription("Width of the text field.");

// currentValue property
PropertyDescriptor currentValue = new PropertyDescriptor( "currentValue", beanClass );
currentValue.setShortDescription("Current value of slider.");

// maximumValue property
PropertyDescriptor maximumValue = new PropertyDescriptor( "maximumValue", beanClass );
maximumValue.setPropertyEditorClass(MaximumValueEditor.class);
maximumValue.setShortDescription("Maximum value of slider.");

// minimumValue property
PropertyDescriptor minimumValue = new PropertyDescriptor( "minimumValue", beanClass );
minimumValue.setPropertyEditorClass(MinimumValueEditor.class);
minimumValue.setShortDescription("Minimum value of slider.");

// ensure PropertyChangeEvent occurs for this property
currentValue.setBound(true);
PropertyDescriptor descriptors[] = { fieldWidth, currentValue, maximumValue, minimumValue };
return descriptors;

// throw RuntimeException if IntrospectionException thrown
catch ( IntrospectionException exception ) {
    throw new RuntimeException( exception.getMessage() );
}

// get currentValue property index
public int getDefaultPropertyIndex() {
    return 1;
}
// return EventSetDescriptors array
public EventSetDescriptor[] getEventSetDescriptors()
    throws RuntimeException
{
    // create array of EventSetDescriptors
    try {
        EventSetDescriptor changed = new
            EventSetDescriptor( beanClass, "propertyChange",
                java.beans.PropertyChangeListener.class,
                "propertyChange" );

        // set event description and name
        changed.setShortDescription(
            "Property change event for currentValue." );
        changed.setDisplayName(
            "SliderFieldPanel value changed" );

        EventSetDescriptor[] descriptors = { changed };

        return descriptors;
    }

    // throw RuntimeException if IntrospectionException
    // thrown
    catch ( IntrospectionException exception ) {
        throw new RuntimeException( exception.getMessage() );
    }
}

// get PropertyChangeEvent index
public int getDefaultEventIndex()
{
    return 0;
}

}  // end class SliderFieldPanelBeanInfo

Fig. 6.56  SliderFieldPanelBeanInfo exposes properties and events for
SliderFieldPanel (part 5 of 5).

Fig. 6.57  Properties and events exposed by SliderFieldPanelBeanInfo.
Every **BeanInfo** class must implement interface **BeanInfo**. This interface describes the methods used by builder tools to determine the exposed features of the bean described by its corresponding **BeanInfo** class. As a convenience, the `java.beans` package includes class **SimpleBeanInfo**, which provides a default implementation of every method in interface **BeanInfo**. The programmer can extend this class and selectively override its methods to implement a proper **BeanInfo** class. Class **SliderFieldPanelBeanInfo** extends class **SimpleBeanInfo** (line 13). In Fig. 6.56, we override **BeanInfo** methods **getBeanDescriptor**, **getIcon**, **getMethodDescriptors**, **getPropertyDescriptors**, **getDefaultValueIndex**, **getEventSetDescriptors** and **getDefaultEventIndex**.

**Software Engineering Observation 6.17**

Class **SimpleBeanInfo** provides a default implementation of every method in interface **BeanInfo**. The programmer can selectively override methods of this class to implement a proper **BeanInfo** class.

Lines 19–28 override method **getBeanDescriptor**, to return a **BeanDescriptor** object. The constructor for **BeanDescriptor** takes as arguments the JavaBean customizer **Class** object. A customizer provides a specialized user interface for customizing a bean. We discuss customizers and specifically the **SliderFieldPanelCustomizer** in Section 6.8.2. Methods **setDisplay** (line 23) and **setShortDescription** (line 24) set the JavaBean’s name and a short description, respectively. The builder tool extracts this information and displays it when selecting the bean.

Lines 31–58 override method **getIcon**. Interface **BeanInfo** defines the constants used by the **switch** statement. The **switch** (lines 35–55) loads the appropriate **Image**. The builder tool uses this **Image** as an icon in the **Component Palette**.

**Software Engineering Observation 6.18**

Method **getIcon** allows a programmer to customize the look of a JavaBean within a builder tool. Common icons are company logos and descriptive graphics.

Lines 62–98 override method **getMethodDescriptors** to return an array of **MethodDescriptor** objects for the **SliderFieldPanel** bean. Each **MethodDescriptor** represents a specific method exposed to the builder tool. Method **getMethodDescriptors** describes the get methods for properties **maximumValue**, **minimumValue**, **currentValue** and **fieldWidth**. The method calls in lines 66–80 may throw **NoSuchMethodException** and **SecurityException** exceptions.

Lines 101–149 override method **getPropertyDescriptors**, to return an array of **PropertyDescriptor** objects for **SliderFieldPanel** properties. Each **PropertyDescriptor** indicates a specific property that should be exposed by a builder tool. There are several ways to construct a **PropertyDescriptor**. In this example, each **PropertyDescriptor** constructor call has the form

```
new PropertyDescriptor( "propertyName", beanClass );
```

where **propertyName** is a **String** that specifies the name of a property defined by the pair of methods **setPropertyName** and **getPropertyName**. Note that the **propertyName** begins with a lowercase letter and the get/set property methods begin the property name with an uppercase letter. We defined **PropertyDescriptor**s for **fieldWidth**, **currentValue**, **minimumValue** and **maximumValue** in class **SliderFieldPanel**.
Method `setShortDescription` sets a short text description of the property. For the `maximumValue` and `minimumValue` properties, we also specify `PropertyEditors` with method `setPropertyEditorClass`. A `PropertyEditor` defines a custom user interface for editing a bean property. We discuss `PropertyEditors`—and specifically `MinimumValueEditor` and `MaximumValueEditor`—in Section 6.8.1. Line 136 specifies that property `currentValue` is a bound property. Some builder tools visually treat bound-property events separately from other events.

**Software Engineering Observation 6.19**

If the set/get methods for a property do not use the JavaBean’s naming convention for properties, there are two other `PropertyDescriptor` constructors in which the actual method names are passed. This allows the builder tools to use nonstandard property methods to expose a property for manipulation by the programmer at design time. This is particularly useful in retrofitting a class as a JavaBean when that class was not originally designed and implemented using JavaBeans design patterns.

Lines 138–139 create the `PropertyDescriptor` array that method `getPropertyDescriptors` returns (line 141). Note the exception handler for `IntrospectionException`. If a `PropertyDescriptor` constructor is unable to confirm the property in the corresponding `Class` object that represents the class definition, the constructor throws an `IntrospectionException`. Because the `BeanInfo` class and its methods are actually used by the builder tool at design time (i.e., during the development of the program in the IDE), the `RuntimeException` thrown in the `catch` handler would normally be caught by the builder tool.

Lines 152–155 define method `getDefaultPropertyIndex` to return the value 1, indicating that the property at position 1 in the `PropertyDescriptor` array returned from `getPropertyDescriptors` is the default property for developers to customize in a builder tool. Typically, the default property is selected when you click a bean. In this example, property `currentValue` is the default property.

Lines 158–184 override method `getEventSetDescriptors` to return an array of `EventSetDescriptor` objects that describes to a builder tool the events supported by this bean. Lines 163–166 define an `EventSetDescriptor` object for the `PropertyChangeEvent` associated with the bound property `currentValue`. The four arguments to the constructor describe the event that should be exposed by the builder tool. The first argument is the `Class` object (beanClass) representing the event source (i.e., the bean that generates the event). The second argument is a `String` representing the event set name (e.g., the mouse event set includes `mousePressed`, `mouseClicked`, `mouseReleased`, `mouseEntered` and `mouseExited`). In this example, the event set name is `propertyChange`. The third argument is the `Class` object representing the event listener interface implemented by listeners for this event. Finally, the last argument is a `String` representing the name of the listener method to call (`propertyChange`) when this event occurs.

When using the standard JavaBeans design patterns, the event set name is part of all the data type names and method names used to process the event. For example, the types and methods for the `propertyChange` event set are: `PropertyChangeListener` (the interface an object must implement to be notified of an event in this event set), `PropertyChangeEvent` (the type passed to a listener method for an event in this event set), `addPropertyChangeListener` (the method called to add a listener for an event in
this event set), `removePropertyChangeListener` (the method called to remove a
listener for an event in this event set) and `firePropertyChange` (the method called to
notify listeners when an event in this event set occurs—this method is named as such by
convention).

**Software Engineering Observation 6.20**

`EventSetDescriptor`s can be constructed with other arguments to expose events that
do not follow the standard JavaBeans design patterns.

A benefit of an `EventSetDescriptor` is customizing the name for the event set
for display in the builder tool. Lines 171–172 call method `setDisplayNames` on the
`EventSetDescriptor` to indicate that its display name should be “`SliderField-
Panel value changed`.”

**Good Programming Practice 6.1**

Customizing the event set name displayed by a builder tool can make the purpose of that
event set more understandable to the component assembler using the bean.

Lines 174–176 create the `EventSetDescriptor` array and return it. Note the
exception handler for `IntrospectionExceptions` at line 182. If an `EventSetDe-
scriptor` constructor is unable to confirm the event in the corresponding `Class` object
that represents the class definition, the constructor throws an `Introspection-
Exception`.

Lines 187–190 define method `getDefaultEventIndex` to return the value 0,
indicating that the property at position 0 in the `EventSetDescriptor` array returned
from `getEventSetDescriptors` is the default event for developers to customize in a
builder tool. Typically, the default event is automatically selected when you click a bean.
In this example, event `propertyChange` is the default event.

### 6.8.1 PropertyEditors

JavaBeans can be customized further by implementing other support classes in addition to
`BeanInfo`. A `PropertyEditor` determines how a particular property is edited inside
a builder tool. The `PropertyEditor` constrains the property to a particular range of val-
ues determined by the programmer. This prevents illegal values that would cause undesired
operation in the JavaBean. A `PropertyEditor` appears often in the property sheet as a
pull-down menu with a list of values. A class implementing interface `PropertyEditor`
is written for every property of a JavaBean that will have a `PropertyEditor`.

**Software Engineering Observation 6.21**

By convention, a `PropertyEditor` class has the same name as the property or type and
ends with `Editor`.

The `PropertyEditors` `MaximumValueEditor` (Fig. 6.58) and `Minimum-
ValueEditor` (Fig. 6.59) edit the `maximumValue` and `minimumValue` properties of
class `SliderFieldPanel`. Both `PropertyEditors` extend class `PropertyEdi-
torSupport`. `PropertyEditorSupport` is a simple implementation of interface
`PropertyEditor`. The `PropertyEditors` each use a combo box with values from
50 through 500, in increments of 50. These values provide a wide range of animation
speeds and prevent an illegal value from being entered for the properties.
```java
// Fig. 6.58 MaximumValueEditor.java
// MaximumValueEditor is the PropertyEditor for the
// maximumValue property of the SliderFieldPanel bean.
package com.deitel.advjhtp1.beans;

// Java core packages
import java.beans.*;

public class MaximumValueEditor extends PropertyEditorSupport {
    private Integer maximumValue;

    // set maximumValue property
    public void setValue( Object value )
    {
        maximumValue = ( Integer ) value;
        firePropertyChange();
    }

    // get maximumValue property
    public Object getValue()
    {
        return maximumValue;
    }

    // set maximumValue property from text string
    public void setAsText( String string )
    {
        // decode may throw NumberFormatException
        try {
            maximumValue = Integer.decode( string );
            firePropertyChange();
        }
        // throw IllegalArgumentException if decode throws
        catch ( NumberFormatException numberFormatException ) {
            throw new IllegalArgumentException();
        }
    }

    // get String array for pull-down menu
    public String[] getTags()
    {
        return new String[] { "50", "100", "150", "200", "250",
                             "300", "350", "400", "450", "500" };}

    // get maximumValue property as string
    public String getAsText()
    {
```

Fig. 6.58  MaximumValueEditor is a PropertyEditor for
SliderFieldPanel’s maximumValue property (part 1 of 2).
Integer `maximumValue` (line 11) stores the value selected in the combo box. Method `setValue` (lines 14–18) takes an `Object value` as a parameter, sets `maximumValue` to `value` and calls method `firePropertyChange`. Method `getValue` (lines 21–24) returns `maximumValue`. Method `setAsText` (lines 27–41) takes a parameter `string` and decodes `string` into a new `Integer` value for `maximumValue`. Method `getTags` (lines 44–48) returns a `String` array of all of the tags that appear in the combo box. Method `getAsText` (lines 51–54) returns `maximumValue` as a `String`. Method `getJavaInitializationString` (lines 57–60) returns `maximumValue` as a `String` for use as a method parameter in generated source code.

`MinimumValueEditor`’s source code (Fig. 6.59) follows the exact same structure as `MaximumValueEditor`, but edits the `minimumValue` property of bean `SliderFieldPanel`.

```
53         return getValue().toString();
54     }
55
56     // get initialization string for Java code
57     public String getJavaInitializationString()
58     {
59         return getValue().toString();
60     }
61
62 }  // end class MaximumValueEditor

Fig. 6.58 MaximumValueEditor is a PropertyEditor for SliderFieldPanel’s maximumValue property (part 2 of 2).

Integer `maximumValue` (line 11) stores the value selected in the combo box. Method `setValue` (lines 14–18) takes an `Object value` as a parameter, sets `maximumValue` to `value` and calls method `firePropertyChange`. Method `getValue` (lines 21–24) returns `maximumValue`. Method `setAsText` (lines 27–41) takes a parameter `string` and decodes `string` into a new `Integer` value for `maximumValue`. Method `getTags` (lines 44–48) returns a `String` array of all of the tags that appear in the combo box. Method `getAsText` (lines 51–54) returns `maximumValue` as a `String`. Method `getJavaInitializationString` (lines 57–60) returns `maximumValue` as a `String` for use as a method parameter in generated source code.

`MinimumValueEditor`’s source code (Fig. 6.59) follows the exact same structure as `MaximumValueEditor`, but edits the `minimumValue` property of bean `SliderFieldPanel`.

```
1     // Fig. 6.59 MinimumValueEditor.java
2     // MinimumValueEditor is the PropertyEditor for the
3     // minimumValue property of the SliderFieldPanel bean.
4     package com.deitel.advjhtp1.beans;
5     
6     // Java core packages
7     import java.beans.*;
8     
9     public class MinimumValueEditor extends PropertyEditorSupport {
10         
11             protected Integer minimumValue;
12         
13             // set minimumValue property
14             public void setValue( Object value )
15             {
16                 minimumValue = ( Integer ) value;
17                 firePropertyChange();
18             }
19         
20             // get value of property minimum
21             public Object getValue()
22             {

Fig. 6.59 MinimumValueEditor is a PropertyEditor for SliderFieldPanel’s minimumValue property (part 1 of 2).
MaximumValueEditor and MinimumValueEditor must be compiled and packaged in the same JAR file as SliderFieldPanel and SliderFieldPanel-BeanInfo. When SliderFieldPanel is installed, the builder tool instantiates the PropertyEditors. MaximumValueEditor and MinimumValueEditor are presented as pull-down menus with values from 50–500 (Fig. 6.60). Try changing the values of maximumValue and minimumValue with the pull-down menus. When the SliderFieldPanel is linked to LogoAnimator2 and executed, the selected values of minimumValue and maximumValue constrain the animation speed within the range 50–500 (Fig. 6.61).
6.8.2 Customizers

The final option for customizing a JavaBean is a Customizer class. A class that implements interface Customizer creates a customized interface for setting the properties of a JavaBean. This interface is separate from the builder-tool style sheet. A Customizer is useful for manipulating JavaBean properties that cannot be edited in the standard property sheet, such as instance fields in objects that are themselves properties.

SliderFieldPanelCustomizer (Fig. 6.62) implements the Customizer interface. A Customizer must extend a Component and provide a no-argument constructor so the builder tool can instantiate it. The constructor (lines 26–75) initializes the components of the customizer. Object changeSupport (line 29) registers PropertyChangeListeners with class PropertyChangeSupport. SliderFieldPanelCustomizer consists of two JLabels and two JComboBoxes. The JLabels label

1. Due to a problem in Forte that prevents bean Customizers from working properly, we use NetBeans 3.2 to demonstrate the example in this section. NetBeans is the open-source development environment project on which Forte is based. NetBeans can be downloaded free of charge from www.netbeans.org.
the JComboBoxes for the minimumValue and maximumValue properties (lines 32–33). Properties minimumValue and maximumValue can be set from 50 through 500, in increments of 50 (lines 36–39). Lines 42–69 add ActionListeners to the JComboBoxes. Selecting a value causes the actionPerformed method of the listener to call the right method to change either maximumValue or minimumValue.

```java
// Fig. 6.62 SliderFieldPanelCustomizer.java
// SliderFieldPanelCustomizer is the Customizer class for
// SliderFieldPanel.
package com.deitel.advjhtp1.beans;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

public class SliderFieldPanelCustomizer extends JPanel
    implements Customizer {
    private JComboBox maximumCombo, minimumCombo;
    private JLabel minimumLabel, maximumLabel;
    protected SliderFieldPanel slider;
    private PropertyChangeSupport changeSupport;
    private static final String[] VALUES = { "50", "100", "150", "200", "250", "300", "350", "400", "450", "500" };

    // initialize GUI components
    public SliderFieldPanelCustomizer() {
        // create PropertyChangeSupport to handle PropertyChange
        changeSupport = new PropertyChangeSupport( this );

        // labels for maximum and minimum properties
        minimumLabel = new JLabel( "Minimum Slider Value:" );
        maximumLabel = new JLabel( "Maximum Slider Value:" );

        // combo boxes adjust maximum and minimum properties
        minimumCombo = new JComboBox( VALUES );
        minimumCombo.setSelectedIndex( 0 );
        maximumCombo = new JComboBox( VALUES );
        maximumCombo.setSelectedIndex( 9 );

        // add ActionListener to minimumValue combo box
        minimumCombo.addActionListener( null );
    }
}
```

Fig. 6.62 SliderFieldPanelCustomizer custom GUI for modifying SliderFieldPanel beans (part 1 of 3).
    new ActionListener() { 
    // handle action of minimum combo box
    public void actionPerformed( ActionEvent event )
    {
        setMinimum( minimumCombo.getSelectedIndex() );
    }
    } // end anonymous inner class
}

// add ActionListener to maximumValue combo box
maximumCombo.addActionListener(
    new ActionListener() {
    // handle action of maximum combo box
    public void actionPerformed( ActionEvent event )
    {
        setMaximum( maximumCombo.getSelectedIndex() );
    }
    } // end anonymous inner class

); // end addActionListener

// add PropertyChangeListener with PropertyChangeSupport
public void addPropertyChangeListener( 
    PropertyChangeListener listener )
{
    changeSupport.addPropertyChangeListener( listener );
}

// remove PropertyChangeListener with PropertyChangeSupport
public void removePropertyChangeListener( 
    PropertyChangeListener listener )
{
    changeSupport.removePropertyChangeListener( listener );
}

// set the customized object
public void setObject( Object bean )
{
    slider = ( SliderFieldPanel ) bean;
}

Fig. 6.62 SliderFieldPanelCustomizer custom GUI for modifying SliderFieldPanel beans (part 2 of 3).
Method `setObject` (lines 78–81) takes as an object argument an instance of the JavaBean being customized. Line 80 casts the object reference to `SliderFieldPanel` and assigns it to instance variable `slider`.

Class `SliderFieldPanelCustomizer` provides methods for adding (lines 84–88) and removing (lines 91–95) `PropertyChangeListener` s. The builder tool registers with `SliderFieldPanelCustomizer` when the customizer is instantiated. A `PropertyChangeSupport` object (line 21) maintains the list of active listeners. Methods `setMinimum` (lines 98–106) and `setMaximum` (lines 109–117) call `firePropertyChangeEvent` to change the `minimumValue` and `maximumValue` properties of the JavaBean. Method `firePropertyChangeEvent` creates a new `PropertyChangeEvent` object with the new and old values of the changing property and sends the event to all registered listeners through `changeSupport`.

`SliderFieldPanelCustomizer` must be packaged in the same JAR file as `SliderFieldPanel` and `SliderFieldPanelBeanInfo`. Once it is installed, right click an instance of `SliderFieldPanel` and select `Customize` (Fig. 6.63). The Customizer Dialog opens and contains the `SliderFieldPanelCustomizer` (Fig. 6.64). Select the desired values for `minimumValue` and `maximumValue` and click the `Close` button. The new values take effect when you execute the application.
6.9 Internet and World Wide Web Resources

**java.sun.com/beans**
The JavaBeans Home Page at the Sun Microsystems, Inc., Web site. Here, you can download the Beans Development Kit (BDK) and other bean-related software. Other features of the site include JavaBeans documentation and specifications, a frequently asked questions list, an overview of integrated development environments that support JavaBeans development, training and support, upcoming JavaBeans events, a searchable directory of JavaBeans components, a support area for marketing your JavaBeans and a variety of on-line resources for communicating with other programmers regarding JavaBeans.

**java.sun.com/beans/spec.html**
Visit this site to download the JavaBeans specification.

**java.sun.com/beans/tools.html**
Visit this site for information about JavaBeans-enabled development tools.

**java.sun.com/beans/directory**
Visit this site for a searchable directory of available beans.
SUMMARY

- A JavaBean is a reusable software component that can be manipulated visually in a builder tool.
- JavaBeans (often called beans) allow developers to reap the benefits of rapid application development in Java by assembling predefined software components to create powerful applications and applets.
- Graphical programming and design environments (often called builder tools) that support beans provide programmers with tremendous flexibility by allowing programmers to reuse and integrate existing disparate components that in many cases, were never intended to be used together.
- The component assembler uses well-defined components to create more robust functionality.
- Sun Microsystems’s Forte for Java Community Edition is an integrated development environment that provides a builder tool for assembling JavaBeans.
- A bean must be installed before it can be manipulated in Forte.
- GUI JavaBeans must be added to a Java Container.
- The property sheet in a builder tool displays a component’s properties and allows them to be edited.
- JavaBeans can be connected with events.
- JavaBeans can be saved to disk as serialized objects or as Java Archive files (JAR). Saving a JavaBean in either of these methods allows other builder tools and code to use the JavaBean.
- Most JavaBeans are GUI components intended to be visually manipulated within a builder tool, such as Forte.
- Most Java Swing components, such as JButtons, are JavaBeans.
- By implementing Serializable, a customized JavaBean can be saved and reloaded in a builder tool or a Java application.
- To use a class as a JavaBean, one must first place it in a Java Archive file (JAR file). A JAR file for a JavaBean must contain a manifest file, which describes the JAR file contents. Manifest files contain attributes (called headers) that describe the individual contents of the JAR.
- When a JAR file containing a JavaBean (or a set of JavaBeans) is loaded into an IDE, the IDE looks at the manifest file to determine which of the classes in the JAR represent JavaBeans. These classes are made available to the programmer in a visual manner.
- All JavaBean-aware development environments know to look for the MANIFEST.MF file in the META-INF directory of the JAR file.
- The Java interpreter can execute an application directly from a JAR file if the manifest file specifies which class in the JAR contains method main.
- To execute a JavaBean from its JAR file, launch the Java interpreter with the -jar command-line option as follows:

  java -jar JARFileName.jar

- The command

  jar cfm JARFileName.jar manifest.tmp files

creates a JAR file.
- A read/write property of a bean is defined as a set/get method pair of the form

  public void setPropertyName( DataType value )
  public DataType getPropertyName()

where PropertyName is replaced in each case by the actual property name. These methods are often referred to as a "property set method" and "property get method," respectively.
• If the property is a `boolean` data type, the `set/get` method pair is normally defined as

```java
public void setPropertyName( boolean value )
public boolean isPropertyName()
```

where the `get` method name begins with the word `is` rather than `get`.

• When a builder tool examines a bean, it inspects the bean methods for pairs of `set/get` methods that represent properties (some builder tools also expose read-only and write-only properties). This is a process known as introspection. If an appropriate `set/get` method pair is found during the introspection process, the builder tool exposes that pair of methods as a property in the builder tool’s user interface.

• A bound property causes the JavaBean that owns the property to notify other objects when there is a change in the bound property’s value. This is accomplished using standard Java event-handling features—registered `PropertyChangeListener`s are notified when the property’s value changes. To support this feature, the `java.beans` package provides interface `PropertyChangeListener` so listeners can be configured to receive property-change notifications, class `PropertyChangeEvent` to provide information to a `PropertyChangeListener` about the change in a property’s value and class `PropertyChangeSupport` to provide the listener registration and notification services (i.e., to maintain the list of listeners and notify them when an event occurs).

• To support registration of listeners for changes to a bound property, a bean defines methods `addPropertyChangeListener` and `removePropertyChangeListener`. Each of these methods calls the corresponding method in the `PropertyChangeSupport` object `changeSupport`. This object provides the event notification services when the property value changes.

• When the bound property changes, the registered `PropertyChangeListener`s must be notified of the change. Each bound-property listener is presented with the old and new property values when notified of the change (the values can be `null` if they are not needed). The `PropertyChangeSupport` object’s `firePropertyChange` method notifies each registered `PropertyChangeListener`.

• An indexed property is like a standard property except that the indexed property is an array of primitives or objects. Two `get` and two `set` methods define an indexed property. The `get` methods are of the form

```java
public Datatype[] getPropertyName()
public DatatypegetPropertyName( int index )
```

The first `get` method returns the entire array of an indexed property. The second `get` method returns the item at the array index indicated by the `get` method’s parameter.

• The `set` methods are of the form

```java
public void setPropertyName( Datatype[] data)
public void setPropertyName( int index, Datatype data )
```

The first `set` method sets the indexed property to the value of the argument. The second `set` method sets the item at the indicated array index to the value of the second parameter.

• A JavaBean can generate programmer-defined events. A programmer-defined event, or custom event, provides functionality that standard Java events do not provide. An event class extends `java.util.EventObject` and the listener class extends `java.util.EventListener`.

• Builder tools use Java’s introspection mechanism to expose a JavaBean’s properties, methods and events if the programmer follows the proper JavaBean design patterns (such as the special naming conventions discussed for `set/get` method pairs that define bean properties). Builder tools use the classes and interfaces of package `java.lang.reflect` to perform introspection. For Java-
Beans that do not follow the JavaBean design patterns or for JavaBeans in which the programmer wants to customize the exposed set of properties, methods and events, the programmer can supply a class that implements interface `BeanInfo` (package `java.beans`). The `BeanInfo` class describes to the builder tool how to present the features of the bean to the programmer.

- Every `BeanInfo` class must implement interface `BeanInfo`. This interface describes the methods used by builder tools to determine the exposed features of the bean described by its corresponding `BeanInfo` class. As a convenience, the `java.beans` package includes class `SimpleBeanInfo`, which provides a default implementation of every method in interface `BeanInfo`. The programmer can extend this class and selectively override its methods to implement a proper `BeanInfo` class.

- Override method `getBeanDescriptor` to return a `BeanDescriptor` object. A `BeanDescriptor` specifies a `Customizer` class and information about the JavaBean.

- Override method `getIcon` to specify an icon to represent the bean in a builder tool.

- Override method `getMethodDescriptors` to return an array of `MethodDescriptor` objects. Each `MethodDescriptor` represents a specific method exposed to the builder tool.

- Override method `getPropertyDescriptors` to return an array of `PropertyDescriptor` objects. Each `PropertyDescriptor` indicates a specific property that should be exposed by a builder tool.

- Override method `getEventSetDescriptors` to return an array of `EventSetDescriptor` objects that describes to a builder tool the events supported by a bean.

- A `PropertyEditor` determines how a particular property is edited inside a builder tool. The `PropertyEditor` constrains the property to a particular range of values determined by the programmer. This prevents illegal values that would cause undesired operation in the JavaBean. A `PropertyEditor` may appear in the property sheet as a pull-down menu with a list of values.

- A class that implements interface `Customizer` creates a customized interface for setting the properties of a JavaBean. This interface is separate from the builder-tool style sheet. A `Customizer` is useful for manipulating JavaBean properties that cannot be edited in the standard property sheet, such as instance fields in objects that are themselves properties.

**TERMINOLOGY**

- `actionPerformed` method of interface `ActionListener`
- `Adapter class`
- `Add to Jar` menu item of `Tools` menu
- `addPropertyChangeListener` method of class `PropertyChangeSupport` bean
- `BeanDescriptor` class
- `BeanInfo` interface
- `Beans` tab of the `Component Palette` toolbar
- bound property of a bean
- builder tool
- `Compile` menu item
- component assembler
- `Component Inspector` window
- `Component Palette` toolbar
- connecting beans
- `Connection mode` button
- `Connection Wizard` dialog
- “connect-the-dots” programming
- creating a JAR file
- custom event
- customize a JavaBean
- `Customize` menu item
- `Customizer` interface
default property
deserialize an object
design pattern
event
event adapter class
event hookup
event listener
event set
event source
event target
**EventListener** interface
**EventObject** class
**EventSetDescriptor** class
**EventSetDescriptor** class
**EventSetDescriptor** class
**Explorer** window
**Filesystems** tab of **Explorer** window
**firePropertyChanged** method of class

**PropertyChangeSupport**

**Form** window
Forte for Java Community Edition
**getBeanDescriptor** method of interface **BeanInfo**
**getDefaultValue** method of interface **BeanInfo**
**getDefaultPropertyIndex** method of interface **BeanInfo**
**getEventSetDescriptors** method of interface **BeanInfo**
**getIcon** method of interface **BeanInfo**
**getJavaInitializationString** method of interface **PropertyEditor**
**getMethodDescriptors** method of interface **BeanInfo**
**getPropertyDescriptors** method of interface **BeanInfo**
**getTags** method of interface **PropertyEditor**

Graphical programming and design environment
header
hook up an event
indexed property
**Install New JavaBean...** menu item
Integrated Development Environment (IDE)
introspection
**IntrospectionException** class
**Iterator** class
**jar** (Java Archive File) utility
**.jar** (Java archive) file extension
**JAR Package** dialog
Java Archive file (JAR file)
**java -jar** (execute an application from a JAR)
**java.beans** package
**java.lang.reflect** package
JavaBean
**JavaBean** header
**JButton** icon

**Main-Class** header
manifest file
**manifest.tmp** file
**META-INFO** directory of a JAR file
**Method Call** radio button of **Connection Wizard** dialog
**MethodDescriptor** class
**Name** header
object serialization
**paintComponent** method of class **JPanel**
**Palette Category** dialog
PNG file
property
property **get** method
property **set** method
property sheet
**PropertyChangeEvent** class
**PropertyChangeListener** interface
**PropertyChangeSupport** class
**PropertyDescriptor** class
**PropertyEditor** interface
**PropertyEditorSupport** class
read/write property
**removePropertyChangeListener** method of class **PropertyChangeSupport**
**repaint** method of class **JPanel**
**Running** tab of Forte
**Selection mode** button
**Serializable** interface
serializing a bean
**Set Layout** menu item
set/get method pair
**setAsText** method of interface **PropertyEditor**
**setDisplayAsText** method of class **BeanDescriptor**
**setDisplayAsText** method of class **EventSetDescriptor**
**setObject** method of interface **Customizer**
**setPropertyEditorClass** method of class **PropertyEditors**
**setShortDescription** method of class **BeanDescriptor**
**setShortDescription** method of class **PropertyDescriptor**
**setValue** method of interface **PropertyEditor**
**Source Editor** window
**stop** method of class **Timer**
SELF-REVIEW EXERCISES

6.1 State whether each of the following is true or false. If false, explain why.
   
   a) A JavaBean is a reusable software component.  
   b) Forte for Java is a builder tool.  
   c) JavaBeans cannot generate events.  
   d) A Customizer modifies an individual property of a JavaBean.  
   e) An indexed property represents an array variable.

6.2 Fill in the blanks in each of the following statements:
   
   a) The four windows of the Forte GUI Editing tab are __________, __________, __________, and __________.
   b) A __________ allows the programmer to customize a property’s value.
   c) In Forte, the __________ provides access to the events supported by a bean that is an event source.
   d) JavaBeans should all implement the __________ interface so they can be saved from a builder tool after being customized by the programmer.
   e) All registered __________ are notified when a bound property’s value changes.
   f) A builder tool uses __________ to expose a JavaBean’s properties, methods and events.
   g) An __________ consists of an event class and a listener class.
   h) For JavaBeans that do not follow the JavaBean design pattern, or for JavaBeans in which the programmer wants to customize the exposed set of properties, methods and events, the programmer can supply a __________ class that describes to the builder tool how to present the features of the bean.
   i) A __________ object describes a property that a builder tool should expose.
   j) A __________ object describes an event set that a builder tool should expose.
   k) A __________ is a custom editor for a bean property that appears in a property sheet.

ANSWERS TO SELF-REVIEW EXERCISES

6.1 a) True.  b) True.  c) False. A JavaBean can generate Java events.  d) False. A Customizer can modify any number of JavaBean properties.  e) True.

6.2 a) Explorer, Component Inspector, Form, Source Editor.  b) property sheet.  c) Component Inspector.  d) Serializable.  e) PropertyChangeListener.  f) introspection.  g) event set.  h) BeanInfo.  i) PropertyDescriptor.  j) EventSetDescriptor.  k) PropertyEditor.

EXERCISES

6.3 Try some of the Swing components provided with Forte. Every Swing component is a JavaBean. While using each bean, try the following:

   a) Inspect the properties of each bean and try modifying them.
   b) Inspect the events supported by each bean and try using those events to hook various beans together.

6.4 Modify ColorSliderPanel to provide a mechanism for viewing the selected color. For this purpose, add a JPanel object to the bean. Test your bean in Forte by changing the background color of a JFrame or other Swing component.
6.5 Create a BeanInfo class for the LogoAnimator2 bean that exposes only the background and animationDelay properties. Test your bean in Forte.

6.6 Modify ColorSliderPanel to use a bound property color instead of indexed property redGreenBlue and custom event ColorEvent. Bound property color is an instance of class Color. Test the bean in Forte by changing the background color of LogoAnimator2.

6.7 Create a Customizer class for ColorSliderPanel that can set a narrower range of values than 0–255 for the red, green and blue sliders. A BeanInfo class must also be created to use this Customizer.

6.8 Modify MaximumValueEditor and MinimumValueEditor to accept the values 1–1000 as a String instead of using a pull-down menu.
Objectives

• To understand the basic concepts of security.
• To understand public-key/private-key cryptography.
• To learn about popular security protocols, such as SSL.
• To understand digital signatures, digital certificates and certification authorities.
• To learn how Java provides solutions to security problems.
• To learn how to produce secure code with Java technology.

*Three may keep a secret, if two of them are dead.*
Benjamin Franklin

*Attack—Repeat—Attack.*
William Frederick Halsey, Jr.

*Private information is practically the source of every large modern fortune.*
Oscar Wilde

*There must be security for all—or not one is safe.*
*The Day the Earth Stood Still*, screenplay by Edmund H. North

*No government can be long secure without formidable opposition.*
Benjamin Disraeli
7.1 Introduction

The explosion of e-business and e-commerce is forcing businesses and consumers to focus on Internet and network security. Consumers are buying products, trading stocks and banking online. They are submitting their credit-card numbers, social-security numbers and other highly confidential information to vendors through Web sites. Businesses are sending confidential information to clients and vendors over the Internet. At the same time, e-businesses are experiencing an increasing number of security attacks. Individuals and organizations are vulnerable to data theft and hacker attacks that can corrupt files and shut down systems, effectively halting business. Hence, security is fundamental.

Modern computer security addresses the various problems and concerns of protecting electronic communications and maintaining network security. There are five fundamental
requirements of a successful, secure transaction: privacy, integrity, authentication, authorization and nonrepudiation. The privacy issue is: How do you ensure that the information you transmit over the Internet has not been captured or passed on to a third party without your knowledge? The integrity issue is: How do you ensure that the information you send or receive has not been compromised or altered? The authentication issue is: How do the sender and receiver of a message prove their identities to each other? The authorization issue is: How do we ensure that users can access certain necessary resources, while valuable information is protected? The nonrepudiation issue is: How do you legally prove that a message was sent or received?

In this chapter, we will explore computer and Java security, from secure electronic transactions to secure coding. The Java programming language provides prevention of and solutions for many of today’s security problems. The Java Sandbox architecture and policy files protect users and systems from malicious programs that would otherwise crash computers or steal valuable information. Several APIs, such as the Java Cryptography Extension (JCE), Java Secure Sockets Extension (JSSE) and the Java Authentication and Authorization Service (JAAS), provide additional security to Java applications.

We encourage you to visit the Web resources provided in Section 7.16 to learn more about the latest developments in e-commerce and Java security. These resources include many informative and entertaining demonstrations.

7.2 Ancient Ciphers to Modern Cryptosystems

The channels through which data pass are inherently unsecure; therefore, any private information passed through these channels must somehow be protected. To secure information, data can be encrypted. Cryptography transforms data by using a cipher, or cryptosystem—a mathematical algorithm for encrypting messages. A key—a string of alpha-numeric characters that acts as a password—is input to the cipher. The cipher uses the key to make data incomprehensible to all but the sender and intended receivers. Unencrypted data is called plaintext; encrypted data is called ciphertext. The algorithm is responsible for encrypting data, while the key acts as a variable—using different keys results in different ciphertext. Only the intended receivers should have the corresponding key to decrypt the ciphertext into plaintext.

Cryptographic ciphers have been used throughout history, first recorded by the ancient Egyptians, to conceal and protect valuable information. In ancient cryptography, messages were encrypted by hand, usually with a method based on the alphabetic characters of the message. The two main types of ciphers were substitution ciphers and transposition ciphers. In a substitution cipher, every occurrence of a given letter is replaced by a different letter; for example, if every “a” is replaced by “b,” every “b” by “c,” etc., the word “security” would encrypt to “tfdvsjuz.” The first prominent substitution cipher was credited to Julius Caesar and is referred to today as the Caesar Cipher. Using the Caesar Cipher, we replace every instance of a letter with the alphabetical letter three positions to the right. For example, according to the Caesar Cipher, the word “security” would encrypt to “vhfxulwb.” In a transposition cipher, the ordering of the letters is shifted; for example, if every other letter, starting with “s,” in the word “security” creates the first word in the ciphertext and the remaining letters create the second word in the ciphertext, the word “security” would encrypt to “scrt euiy.” Complicated ciphers are created by combining substitution and transposition ciphers. For example, with the substitution cipher first, then the transpo-
sition cipher, the word “security” would encrypt to “tdsu fvjz.” The problem with many historical ciphers is that their security relied on the sender and receiver to remember the encryption algorithm and keep it secret. Such algorithms are called restricted algorithms. Restricted algorithms are not feasible to implement among a large group of people. Imagine if the security of U.S. government communications relied on every U.S. government employee to keep a secret; the encryption algorithm could be compromised easily.

Modern cryptosystems are digital. Their algorithms are based on the individual bits or blocks (a group of bits) of a message, rather than letters of the alphabet. Encryption and decryption keys are binary strings with a given key length. For example, 128-bit encryption systems have a key length of 128 bits. Longer keys have stronger encryption; it takes more time and computing power to crack messages encrypted with longer keys.

Until January 2000, the U.S. government placed restrictions on the strength of cryptosystems that could be exported from the United States. Federal regulations limited the key length of encryption algorithms. Today, the regulations on exporting products that employ cryptography are less stringent. Any cryptography product may be exported, as long as the end user is not a foreign government or from a country with embargo restrictions on it.¹

### 7.3 Secret-key Cryptography

In the past, organizations wishing to maintain a secure computing environment used symmetric cryptography, also known as secret-key cryptography. Secret-key cryptography utilizes the same secret key to encrypt and decrypt messages (Fig. 7.1). The sender encrypts a message using the secret key, then sends the encrypted message to the intended recipient. A fundamental problem with secret-key cryptography is that before two people can communicate securely, they must find a secure way to exchange the secret key. One approach is to have the key delivered by a courier, such as a mail service or FedEx. While this approach may be feasible when two individuals communicate, it is not efficient for securing communication in a large network, nor can it be considered completely secure. The privacy and the integrity of the message could be compromised if the key is intercepted as it is passed between the sender and the receiver over unsecure channels. Also, since both parties in the transaction use the same key to encipher and decipher a message, one cannot authenticate which party created the message. Finally, to keep communications with each receiver private, a sender needs a different secret key for each receiver. As a result, organizations may have huge numbers of secret keys to maintain.

An alternative approach to the key-exchange problem is to have a central authority, called a key distribution center (KDC). The key distribution center shares a (different) secret key with every user in the network. In this system, the key distribution center generates a session key to be used for a transaction (Fig. 7.2). Next, the key distribution center distributes the session key to the sender and receiver, encrypting the session key itself with the secret key they each share with the key distribution center. For example, suppose a merchant and a customer want to conduct a secure transaction. The merchant and the customer each have unique secret keys they share with the key distribution center. The key distribution center generates a session key for the merchant and customer to use in the transaction. The key distribution center then sends the session key for the transaction to the merchant, encrypted using the secret key the merchant already shares with the center. The key distribution center sends the same session key for the transaction to the customer, encrypted using the secret key the customer already shares with the key distribution center. Once the
merchant and the customer have the session key for the transaction, they can communicate with each other, encrypting their messages using the shared session key.

Using a key distribution center reduces the number of courier deliveries (again, by means such as mail) of secret keys to each user in the network. In addition, users can have a new secret key for each communication with other users in the network, which greatly increases the overall security of the network. However, if the security of the key distribution center is compromised, then so is the security of the entire network.

One of the most commonly used symmetric encryption algorithms is the Data Encryption Standard (DES). Horst Feistel of IBM created the Lucifer algorithm, which the United States government and the National Security Agency (NSA) chose as the DES in the 1970s. DES has a key length of 56 bits and encrypts data in 64-bit blocks, a type of encryption known as a block cipher. A block cipher is an encryption method that creates groups of bits from an original message, then applies an encryption algorithm to the block as a whole, rather than as individual bits. This method reduces the amount of computer processing power and time required, while maintaining a fair level of security. For many years, DES was the encryption standard set by the U.S. government and the American National Standards Institute (ANSI). However, due to advances in technology and computing speed, DES is no longer considered secure. In the late 1990s, specialized DES cracker machines were built that recovered DES keys after just several hours. As a result, the old standard of symmetric encryption has been replaced by Triple DES, or 3DES, a variant of DES that is essentially three DES systems in series, each having its own secret key. 3DES is more secure, however the three passes through the DES algorithm result in slower performance. The United States government recently selected a new, more secure standard for symmetric encryption to replace DES. The new standard is called the Advanced Encryption Standard (AES). The National Institute of Standards and Technology (NIST), which sets the cryptographic standards for the U.S. government, is evaluating Rijndael as the encryption method for AES. Rijndael is a block cipher developed by Dr. Joan Daemen and Dr. Vincent Rijmen of Belgium. Rijndael can be used with key sizes and block sizes of 128, 192 or 256 bits. Rijndael was chosen over four other finalists as the AES candidate because of its high security, performance, efficiency, flexibility and low-memory requirement for computing systems. For more information about AES, visit csrc.nist.gov/encryption/aes.

### 7.4 Public-key Cryptography

In 1976, Whitfield Diffie and Martin Hellman, researchers at Stanford University, developed public-key cryptography to solve the problem of exchanging keys securely. Public-key cryptography is asymmetric. It uses two inversely related keys: A public key and a private key. The private key is kept secret by its owner, while the public key is freely distributed. If the public key is used to encrypt a message, only the corresponding private key can decrypt it, and vice versa (Fig. 7.3). Each party in a transaction has both a public key and a private key. To transmit a message securely, the sender uses the receiver’s public key to encrypt the message. The receiver then decrypts the message using his or her unique private key. Assuming that the private key has been kept secret, the message cannot be read by anyone other than the intended receiver; the system ensures the privacy of the message. The defining property of a secure public-key algorithm is that it is computationally infeasible to deduce the private key from the public key. Although the two keys are mathematically related, deriving one from the other would take enormous amounts of computing power and
time, enough to discourage attempts to deduce the private key. An outside party cannot participate in communication without the correct keys. The security of the entire process is based on the secrecy of the private keys. Therefore, if a third party does obtain the decryption key, the security of the whole system is compromised. If a system’s integrity is compromised, the user can simply change the key, instead of changing the entire encryption or decryption algorithm.

**Fig. 7.1** Encrypting and decrypting a message using a symmetric secret key.

**Fig. 7.2** Distributing a session key with a key distribution center.
Either the public key or the private key can be used to encrypt or decrypt messages. For example, if a customer uses a merchant’s public key to encrypt a message, only the merchant can decrypt the message, using the merchant’s private key. Thus, the merchant’s identity can be authenticated, since only the merchant knows the private key. However, the merchant has no way of validating the customer’s identity, since the encryption key the customer used is publicly available. If the encryption key is the sender’s private key and the decryption key is the sender’s public key, the sender of the message can be authenticated. For example, suppose a customer sends a merchant a message encrypted using the customer’s private key. The merchant decrypts the message using the customer’s public key. Since the customer encrypted the message using his or her private key, the merchant can be confident of the customer’s identity. This process provides for authentication of the sender, not confidentiality, as anyone could decrypt the message with the sender’s public key. This system works as long as the merchant can be sure that the public key with which the merchant decrypted the message belongs to the customer, and not a third party posing as the customer. The problem of proving ownership of a public key is discussed in Section 7.10.

These two methods of public-key encryption can be used together to authenticate both participants in a communication (Fig. 7.4). Suppose a merchant wants to send a message securely to a customer so that only the customer can read it, and suppose also that the merchant wants to provide proof to the customer that the merchant (not an unknown third party) actually sent the message. First, the merchant encrypts the message using the customer's public key. This step guarantees that only the customer can read the message. Then the merchant encrypts the result using the merchant’s private key, which proves the identity of the merchant. The customer decrypts the message in reverse order. First, the customer uses the merchant’s public key. Since only the merchant could have encrypted the message with the inversely related private key, this step authenticates the merchant. Then the customer uses the customer’s private key to decrypt the next level of encryption. This step ensures that the message content was kept private in the transmission, since only the customer has the key to decrypt the message. This system provides for extremely secure transactions; however, the cost and time necessary for setting up such a system prevent this system from present use.

The most commonly used public-key algorithm is RSA, an encryption system developed in 1977 by MIT professors Ron Rivest, Adi Shamir and Leonard Adleman. Today, their encryption and authentication technologies are used by most Fortune 1000 companies and leading e-commerce businesses. With the emergence of the Internet and the World Wide Web, their security work has become even more significant and plays a crucial role in e-commerce transactions. Their encryption products are built into hundreds of millions of copies of the most popular Internet applications, including Web browsers, commerce servers and e-mail systems. Most secure e-commerce transactions and communications on the Internet use RSA products. (For more information about RSA, cryptography and security, visit www.rsasecurity.com).

Pretty Good Privacy (PGP) is a public-key encryption system used for encrypting e-mail messages and files. PGP was designed in 1991 by Phillip Zimmermann. PGP also provides digital signatures (see Section 7.9, Digital Signatures) that confirm the author of an e-mail or public posting. PGP is based on a “web of trust”; each client in a network can vouch for another client’s identity to prove ownership of a public key. Clients use the “web of trust” to authenticate one another. To learn more about PGP and to download a free copy of the software, go to the MIT distribution center for PGP at web.mit.edu/network/pgp.html.
7.5 Cryptanalysis

Even if keys are kept secret, it may be possible to compromise the security of a system. Trying to decrypt ciphertext without knowledge of the decryption key is known as cryptanalysis. Commercial encryption systems are constantly being researched by cryptologists to ensure that the systems are not vulnerable to a cryptanalytic attack. The most common form of cryptanalytic attacks are those in which the encryption algorithm is analyzed to find relations between bits of the encryption key and bits of the ciphertext. Often, these relations are only statistical in nature and incorporate an analyzer’s outside knowledge about the plaintext. The goal of such an attack is to determine the key from the ciphertext.

Weak statistical trends between ciphertext and keys can be exploited to gain knowledge about the key if enough ciphertext is known. Proper key management and expiration dates on keys help prevent cryptanalytic attacks. When a key is used for long periods of time, more ciphertext is generated that can be beneficial to an attacker trying to derive a key. If a key is unknowingly recovered by an attacker, it can be used to decrypt every message for the life of that key. Using public-key cryptography to exchange secret keys securely allows a new secret key to encrypt every message.

7.6 Key Agreement Protocols

A drawback of public-key algorithms is that they are not efficient for sending large amounts of data. They require significant computer power, which slows down communication. Public-key algorithms should not be thought of as a replacement for secret-key algorithms. Instead, public-key algorithms can be used to allow two parties to agree upon a key to be used for secret-key encryption over an unsecure medium. The process by which two parties can exchange keys over an unsecure medium is called a key agreement protocol. A protocol sets the rules for communication: Exactly what encryption algorithm(s) is (are) going to be used?

Fig. 7.3 Encrypting and decrypting a message using public-key cryptography.
The most common key agreement protocol is a digital envelope (Fig. 7.5). With a digital envelope, the message is encrypted using a secret key (Step 1), and the secret key is encrypted using public-key encryption (Step 2). The sender attaches the encrypted secret key to the encrypted message and sends the receiver the entire package. The sender could also digitally sign the package before sending it to prove the sender’s identity to the receiver (Section 7.9). To decrypt the package, the receiver first decrypts the secret key, using the receiver’s private key. Then the receiver uses the secret key to decrypt the actual message. Since only the receiver can decrypt the encrypted secret key, the sender can be sure that only the intended receiver is reading the message.

**7.7 Key Management**

Maintaining the secrecy of private keys is crucial to keeping cryptographic systems secure. Most compromises in security result from poor key management (e.g., the mishandling of private keys, resulting in key theft) rather than attacks that attempt to guess the keys. A main component of key management is key generation—the process by which keys are created. A malicious third party could try to decrypt a message by using every possible decryption key, a process known as brute-force cracking. Key-generation algorithms are sometimes unintentionally constructed to choose from only a small subset of possible keys. If the subset is too small, then the encrypted data is more susceptible to brute-force attacks. Therefore, it is important to have a key-generation program that can generate a large number of keys as randomly as possible. Keys are made more secure by choosing a key length so large that it is computationally infeasible to try all combinations.
7.8 Java Cryptography Extension (JCE)

The Java Cryptography Extension (JCE) provides Java applications with several security facilities. JCE supports secret-key encryption, such as 3DES, and public-key algorithms, such as Diffie-Hellman and RSA. Customizable levels of security are available through multiple encryption algorithms and various key sizes. The JCE architecture is provider-based—developers can add new algorithms to their programs by adding new algorithm providers. Each provider may support many different algorithms. This feature allows developers to use their own algorithms from trusted sources with the JCE API (the API is located at <java.sun.com/products/jce/doc/guide/API_users_guide.html>).

7.8.1 Password-Based Encoding with JCE

Class EncipherDecipher (Fig. 7.6) uses JCE to demonstrate Password-Based Encryption (PBE). Class EncipherDecipher provides users with a graphical user interface that allows them to specify the file name of a file that the application will use to write to and read from, the contents of the file to encrypt/decrypt, and the password used to encrypt/decrypt the file. The password-based encryption algorithm implementation uses an array of bytes (lines 28–31)—called a salt—and an integer (line 34) to randomize the sets of generated keys.

Fig. 7.5 Creating a digital envelope.
Line 45 adds a security provider implementation to the JVM. Each system must set a security provider implementation. The security provider implementation provides the various algorithm implementations that clients can use when selecting encrypting and decrypting techniques.

Constructor `EncipherDecipher` (lines 42–130) creates a `JFrame` that contains three panels. The top panel (lines 52–73) contains two labels and two textfields that allow the user to input the file name and the encryption password to use. Lines 76–88 create the middle panel which allows the user to write the contents that the application will encrypt and write to the file. Lines 91–121 create the bottom panel, which contains two buttons, button `Encrypt and Write to File` and button `Read from File and Decrypt`. When a user presses button `Encrypt and Write to File`, lines 100–103 handle the event by invoking method `encryptAndWriteToFile` (lines 131–277). When a user presses button `Read from File and Decrypt`, lines 115–118 handle the event by invoking method `readFromFileAndDecrypt` (lines 280–384).

```
// EncipherDecipher.java
// Displays a frame that allows users to specify
// a password and a file name. Contents written
// to an Editor Pane can be encrypted and written
// to a file, or encrypted contents can be read from
// a file and decrypted
package com.deitel.advjhtp1.security.jce;

// Java core package
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.spec.*;

// third-party packages
import com.sun.crypto.provider.SunJCE;

// Java extension package
import javax.swing.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class EncipherDecipher extends JFrame {

    // salt for password-based encryption-decryption algorithm
    private static final byte[] salt = {
        ( byte )0xf5, ( byte )0x33, ( byte )0x01, ( byte )0x2a,
        ( byte )0xb2, ( byte )0xcc, ( byte )0xe4, ( byte )0x7f
    };

    // iteration count
    private int iterationCount = 100;

Fig. 7.6  EncipherDecipher application for demonstrating Password-Based Encryption (part 1 of 8).
// user input components.
private JTextField passwordTextField;
private JTextField fileNameTextField;
private JEditorPane fileContentsEditorPane;

// frame constructor
public EncipherDecipher() {

    // set security provider
    Security.addProvider( new SunJCE() );

    // initialize main frame
    setSize( new Dimension( 400, 400 ) );
    setTitle( "Encryption and Decryption Example" );

    // construct top panel
    JPanel topPanel = new JPanel();
    topPanel.setBorder( BorderFactory.createLineBorder( Color.black ) );
    topPanel.setLayout( new BorderLayout() );

    // panel where password and file name labels will be placed
    JPanel labelsPanel = new JPanel();
    labelsPanel.setLayout( new GridLayout( 2, 1 ) );
    JLabel passwordLabel = new JLabel( " Password: " );
    JLabel fileNameLabel = new JLabel( " File Name: " );
    labelsPanel.add( fileNameLabel );
    labelsPanel.add( passwordLabel );
    topPanel.add( labelsPanel, BorderLayout.WEST );

    // panel where password and file name textfields will be placed
    JPanel textFieldsPanel = new JPanel();
    textFieldsPanel.setLayout( new GridLayout( 2, 1 ) );
    passwordTextField = new JPasswordField();
    fileNameTextField = new JTextField();
    textFieldsPanel.add( fileNameTextField );
    textFieldsPanel.add( passwordTextField );
    topPanel.add( textFieldsPanel, BorderLayout.CENTER );

    // construct middle panel
    JPanel middlePanel = new JPanel();
    middlePanel.setLayout( new BorderLayout() );

    // construct and place title label for contents pane
    JLabel fileContentsLabel = new JLabel();
    fileContentsLabel.setText( " File Contents" );
    middlePanel.add( fileContentsLabel, BorderLayout.NORTH );

    // initialize and place editor pane within scroll panel
    fileContentsEditorPane = new JEditorPane();

Fig. 7.6 EncipherDecipher application for demonstrating Password-Based Encryption (part 2 of 8).
middlePanel.add(
    new JScrollPane( fileContentsEditorPane ),
    BorderLayout.CENTER);

// construct bottom panel
JPanel bottomPanel = new JPanel();

// create encrypt button
JButton encryptButton =
    new JButton( "Encrypt and Write to File" );
encryptButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            encryptAndWriteToFile();
        }
    });
bottomPanel.add( encryptButton );

// create decrypt button
JButton decryptButton =
    new JButton( "Read from File and Decrypt" );
decryptButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            readFromFileAndDecrypt();
        }
    });
bottomPanel.add( decryptButton );

// initialize main frame window
JPanel contentPane = ( JPanel ) this.getContentPane();
contentPane.setLayout( new BorderLayout() );
contentPane.add( topPanel, BorderLayout.NORTH );
contentPane.add( middlePanel, BorderLayout.CENTER );
contentPane.add( bottomPanel, BorderLayout.SOUTH );
}

// obtain contents from editor pane and encrypt
private void encryptAndWriteToFile()
{
    // obtain user input
    String originalText = fileContentsEditorPane.getText();
String password = passwordTextField.getText();
String fileName = fileNameTextField.getText();

// create secret key and get cipher instance
Cipher cipher = null;

try {
    // create password based encryption key object
    PBEKeySpec keySpec =
        new PBEKeySpec( password.toCharArray() );

    // obtain instance for secret key factory
    SecretKeyFactory keyFactory =
        SecretKeyFactory.getInstance( "PBEWithMD5AndDES" );

    // generate secret key for encryption
    SecretKey secretKey = keyFactory.generateSecret( keySpec );

    // specifies parameters used with password based encryption
    PBEParameterSpec parameterSpec =
        new PBEParameterSpec( salt, iterationCount );

    // obtain cipher instance reference
    cipher = Cipher.getInstance( "PBEWithMD5AndDES" );

    // initialize cipher in encrypt mode
    cipher.init( Cipher.ENCRYPT_MODE, secretKey, parameterSpec );
}

// handle NoSuchAlgorithmException
catch ( NoSuchAlgorithmException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle InvalidKeySpecException
catch ( InvalidKeySpecException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle InvalidKeyException
catch ( InvalidKeyException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle NoSuchPaddingException
catch ( NoSuchPaddingException exception ) {
    exception.printStackTrace();
}

Fig. 7.6 EncipherDecipher application for demonstrating Password-Based Encryption (part 4 of 8).
```
System.exit( 1 );
}

// handle InvalidAlgorithmParameterException
catch ( InvalidAlgorithmParameterException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// create array of bytes
byte[] outputArray = null;
try {
    outputArray = originalText.getBytes( "ISO-8859-1" );
}

// handle UnsupportedEncodingException
catch ( UnsupportedEncodingException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// create FileOutputStream
File file = new File( fileName );
FileOutputStream fileOutputStream = null;
try {
    fileOutputStream = new FileOutputStream( file );
}

// handle IOException
catch ( IOException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// create CipherOutputStream
CipherOutputStream out =
    new CipherOutputStream( fileOutputStream, cipher );

// write contents to file and close
try {
    out.write( outputArray );
    out.flush();
    out.close();
}

// handle IOException
catch ( IOException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}
```

Fig. 7.6 EncipherDecipher application for demonstrating Password-Based Encryption (part 5 of 8).
// contain bytes read from file
Vector fileBytes = new Vector();

// read contents from file to show user encrypted text
try {
    FileInputStream in = new FileInputStream(file);

    // read bytes from stream.
    byte contents;

    while (in.available() > 0) {
        contents = (byte)in.read();
        fileBytes.add(new Byte(contents));
    }

    in.close();
}

// handle IOException
catch (IOException exception) {
    exception.printStackTrace();
    System.exit(1);
}

// create byte array from contents in Vector fileBytes
byte[] encryptedText = new byte[fileBytes.size()];

for (int i = 0; i < fileBytes.size(); i++) {
    encryptedText[i] = ((Byte)fileBytes.elementAt(i)).byteValue();
}

// update Editor Pane contents
fileContentsEditorPane.setText(new String(encryptedText));

// obtain contents from file and decrypt
private void readFromFileAndDecrypt()
{

    // used to rebuild byte list
    Vector fileBytes = new Vector();

    // obtain user input
    String password = passwordTextField.getText();
    String fileName = fileNameTextField.getText();

    // create secret key
    Cipher cipher = null;
try {
    // create password based encryption key object
    PBEKeySpec keySpec =
        new PBEKeySpec( password.toCharArray() );

    // obtain instance for secret key factory
    SecretKeyFactory keyFactory =
        SecretKeyFactory.getInstance( "PBEWithMD5AndDES" );

    // generate secret key for encryption
    SecretKey secretKey = keyFactory.generateSecret( keySpec );

    // specifies parameters used with password based encryption
    PBEParameterSpec parameterSpec =
        new PBEParameterSpec( salt, iterationCount );

    // obtain cipher instance reference.
    cipher = Cipher.getInstance( "PBEWithMD5AndDES" );

    // initialize cipher in decrypt mode
    cipher.init( Cipher.DECRYPT_MODE, secretKey,
                 parameterSpec );
}

// handle NoSuchAlgorithmException
catch ( NoSuchAlgorithmException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle InvalidKeySpecException
catch ( InvalidKeySpecException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle InvalidKeyException
catch ( InvalidKeyException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle NoSuchPaddingException
catch ( NoSuchPaddingException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// handle InvalidAlgorithmParameterException
catch ( InvalidAlgorithmParameterException exception ) {
    exception.printStackTrace();
    System.exit( 1 );
}

// EncipherDecipher application for demonstrating Password-Based Encryption (part 7 of 8).
344    System.exit( 1 );
345 }
346
347    // read and decrypt contents from file
348    try {
349        File file = new File( fileName );
350        FileInputStream fileInputStream =
351            new FileInputStream( file );
352
353        CipherInputStream in =
354            new CipherInputStream( fileInputStream, cipher );
355
356        // read bytes from stream.
357        byte contents = ( byte ) in.read();
358
359        while ( contents != -1 ) {
360            fileBytes.add( new Byte( contents ) );
361            contents = ( byte ) in.read();
362        }
363        in.close();
364    }
365
366    // handle IOException
367    catch ( IOException exception ) {
368        exception.printStackTrace();
369        System.exit( 1 );
370    }
371
372    // create byte array from contents in Vector fileBytes
373    byte[] decryptedText = new byte[ fileBytes.size() ];
374    for ( int i = 0; i < fileBytes.size(); i++ ) {
375        decryptedText[ i ] =
376            ( ( Byte )fileBytes.elementAt( i ) ).byteValue();
377    }
378
379    // update Editor Pane contents.
380    fileContentsEditorPane.setText( new String( decryptedText ) );
381 }
382
383    // create frame and display
384    public static void main( String[] args )
385    {
386        EncipherDecipher crypto =
387            new EncipherDecipher();
388        crypto.validate();
389        crypto.setVisible( true );
390    }
391
392

Fig. 7.6 EncipherDecipher application for demonstrating Password-Based Encryption (part 8 of 8).
Method `encryptAndWriteToFile` (lines 131–277) obtains the user’s input from the both `JTextField`s and the `JEditorPane`. Lines 147–148 create a `PBEKeySpec` instance. The `PBEKeySpec` instance acts as a wrapper for the array of characters that represents the password for encrypting and decrypting an array of bytes. Class `Cipher` is the fundamental building block for applications that use JCE. A `Cipher` performs encryption and decryption using a specified algorithm (e.g., DES, 3DES, Blowfish, etc.). Lines 151–152 obtain a reference to a `SecretKeyFactory`, which generates secret keys. Line 155 generates a `SecretKey` using the `PBEKeySpec` instance from lines 147–148. Lines 158–159 create a `PBEParameterSpec` instance, which contains randomization information such as the `salt` and the `iterationCount`. Line 162 obtains an instance of a `PBEWithMD5AndDES` algorithm `Cipher`. Line 165–166 initializes `Cipher` to encryption mode using the `SecretKey` and the `PBEParameterSpec` instances.

Lines 170–173 handle `NoSuchAlgorithmException`s, which occur when the program specifies a non-existent algorithm. Lines 176–179 handle `InvalidKeySpecException`s, which occur when an invalid key specification is handed to method `generateSecret` from `SecretKeyFactory`. Lines 182–185 handle all `InvalidKeyException`s which occur when an invalid key is handed to method `init` from `Cipher`. Lines 188–191 handle `NoSuchPaddingException`s, which occur when an application specifies an invalid padding scheme. Lines 194–197 handle `InvalidAlgorithmParameterException`s, which method `init` of class `Cipher` throws if an application specifies invalid algorithm parameters.

Line 203 converts the `String` obtained from `JEditorPanel` into an array of bytes. Method `getBytes` ensures that the conversion of a `String` to an array of bytes conforms to the ISO-8859-1 standard. Lines 207–210 catch an `UnsupportedEncodingException` if the application specifies an invalid character encoding standard.

Lines 213–218 instantiate a `FileOutputStream CipherOutputStream` (lines 227–228) acts as the `decorator` in the `Decorator design pattern` (Section 7.8.2) to add encryption capability to the `FileOutputStream` instance. The `CipherOutputStream` encodes bytes using the specified `Cipher` object before writing those bytes to the `FileOutputStream`. Lines 232–234 write the contents to the file, and finalize the operation by closing the file. Lines 244–276 read the newly encoded file contents and display them in the `JEditorPane` so the user can see the encrypted text.


Figure 7.7 displays the contents that application `EncipherDecipher` will encrypt and write to file `TestFile.txt` using password “I am a BIG secret!”. The image on the right displays the contents of the file after pressing button `Encrypt and Write to File`.

For more information on JCE, please visit the JCE Web site at `java.sun.com/jce`. Refer to the included documentation for download and installation instructions.
7.8.2 Decorator Design Pattern

The preceding program uses an important design pattern—the Decorator design pattern. Method `encryptAndWriteToFile` writes encrypted data to a file. However, neither the `CipherOutputStream` nor the `FileOutputStream`, by itself, can encrypt data and write those data to a file. By “chaining” these two objects together—i.e., by passing a `FileOutputStream` reference to the `CipherOutputStream` constructor—the method can encrypt data and write those data to a file. This “chaining” is an example of the Decorator design pattern, which allows an object to gain additional capabilities dynamically. In this example, the `CipherOutputStream` decorates the `FileOutputStream`—the `CipherOutputStream` provides the `FileOutputStream` with the capability to encrypt data before writing those data to a file. One benefit to this pattern is that designers need not create additional classes (e.g., using inheritance) to extend the functionality of a particular class. For example, because a `FileOutputStream` object can gain the behavior of a `CipherOutputStream` dynamically, we need not create a separate class called `CipherFileOutputStream`, which would implement the behaviors of both classes. Lines 227–228 accomplish the same result simply by chaining the streams together.

If necessary, we could extend this principle further and decorate a `CipherOutputStream` (which decorates a `FileOutputStream`) with an `ObjectOutputStream`. The resulting `ObjectOutputStream` instance would enable us to write encrypted objects to a file. Using the Decorator design pattern, we would write

```java
ObjectOutputStream objectStream = new ObjectOutputStream(
    new CipherOutputStream(
        new FileOutputStream( filename ), cipher ) );
```

We can chain objects in this manner because `CipherOutputStream`, `ObjectOutputStream` and `FileOutputStream` extend abstract superclass `OutputStream`, and each subclass constructor takes an `OutputStream` reference as a parameter. If Java’s stream objects did not use the Decorator pattern (i.e., did not satisfy these two requirements), we would have to design classes `CipherFileOutputStream`, `CipherObjectOut-
putStream, CipherObjectFileOutputStream and ObjectFileOutputStream to achieve the same functionality. If we were to chain more objects without using the Decorator pattern, the number of classes would grow exponentially.

### 7.9 Digital Signatures

*Digital signatures*—the electronic equivalent of written signatures—were developed for use in public-key cryptography to solve the problems of authentication and integrity (see Microsoft Authenticode feature). A digital signature authenticates the sender’s identity and, like a written signature, is difficult to forge. To create a digital signature, a sender first takes the original plaintext message and runs it through a *hash function*, which is a mathematical calculation that gives the message a *hash value*. The Secure Hash Algorithm (SHA-1) is the standard for hash functions. Running a message through the SHA-1 algorithm produces a 160-bit hash value. For example, using SHA-1, the phrase “Buy 100 shares of company X” produces the hash value D8 A9 B6 9F 72 65 0B D5 6D 0C 47 00 95 0D FD 31 96 0A FD B5. MD5 is another popular hash function, which was developed by Ronald Rivest to verify data integrity through a 128-bit hash value of the input file. Examples of SHA-1 and MD5 are available at [home.istar.ca/~neutron/messagedigest](http://home.istar.ca/~neutron/messagedigest). At this site, users can input text or files into a program to generate the hash value. The hash value is also known as a *message digest*. The chance that two different messages will have the same message digest is statistically insignificant. *Collision* occurs when multiple messages have the same hash value. It is computationally infeasible to compute a message from its hash value or to find two different messages with the same hash value.

Next, the sender uses its private key to encrypt the message digest. This step creates a digital signature and authenticates the sender, since only the owner of that private key could encrypt the message. The sender encrypts the original message with the receiver’s public key and sends the encrypted message and the digital signature to the receiver. The receiver uses the sender’s public key to decipher the original digital signature and reveal the message digest. The receiver then uses his or her own private key to decipher the original message. Finally, the receiver applies the agreed upon hash function (e.g. SHA-1 or MD5) to the original message. If the hash value of the original message matches the message digest included in the signature, there is *message integrity*—the message has not been altered in transmission.

There is a fundamental difference between digital signatures and handwritten signatures. A handwritten signature is independent of the document being signed. Thus, if someone can forge a handwritten signature, he or she can use that signature to forge multiple documents. A digital signature is created using the contents of the document. Therefore, your digital signature is different for each document you sign.

Digital signatures do not provide proof that a message has been sent. Consider the following situation: A contractor sends a company a digitally signed contract, which the contractor later would like to revoke. The contractor could do so by releasing the private key and claiming that the digitally signed contract came from an intruder who stole the contractor’s private key. *Timestamping*, which binds a time and date to a digital document, can help solve the problem of non-repudiation. For example, suppose the company and the contractor are negotiating a contract. The company requires the contractor to digitally sign the contract, and have the document digitally time-stamped by a third party called a *times-tamping agency*. The contractor sends the digitally signed contract to the time-stamping agency. The privacy of the message is maintained, since the timestamping agency sees only
the encrypted, digitally signed message (as opposed to the original plaintext message). The
timestamping agency affixes the time and date of receipt to the encrypted, signed message
and digitally signs the whole package with the timestamping agency’s private key. The
timestamp cannot be altered by anyone except the timestamping agency, since no one else
possesses the timestamping agency’s private key. Unless the contractor reports the private
key to have been compromised before the document was timestamped, the contractor
cannot legally prove that the document was signed by an unauthorized third party. The
sender could also require the receiver to sign and timestamp the message digitally as proof
of receipt. To learn more about timestamping, visit AuthentiDate.com.

The U.S. government’s digital-authentication standard is called the Digital Signature
Algorithm (DSA). The U.S. government recently passed legislation that makes digital sig-
natures as legally binding as handwritten signatures. This legislation will result in an
increase in e-business. For the latest news about U.S. government legislation in information
security, visit www.itaa.org/infosec. For more information about the bills, visit the
following government sites:

thomas.loc.gov/cgi-bin/bdquery/z?d106:hr.01714:
thomas.loc.gov/cgi-bin/bdquery/z?d106:s.00761:

7.10 Public-key Infrastructure, Certificates and Certification
Authorities

One problem with public-key cryptography is that anyone with a set of keys could assume
another party’s identity. For example, say a customer wants to place an order with an online
merchant. How does the customer know that the Web site indeed belongs to that merchant
and not to a third party that posted a site and is masquerading as a merchant to steal credit-
card information? Public-Key Infrastructure (PKI) integrates public-key cryptography
with digital certificates and certificate authorities to authenticate parties in a transaction.

A digital certificate is a digital document that identifies a user and is issued by a cer-
tificate authority (CA). A digital certificate includes the name of the subject (the company
or individual being certified), the subject’s public key, a serial number, an expiration date,
the signature of the trusted certificate authority and any other relevant information
(Fig. 7.8). A CA is a financial institution or other trusted third party, such as VeriSign. Once
issued, digital certificates are publicly available and are held by the certificate authority in
certificate repositories.

The CA signs the certificate by encrypting either the subject’s public key or a hash value
of the public key using the CA’s own private key. The CA has to verify every subject’s public
key. Thus, users must trust the public key of a CA. Usually, each CA is part of a certificate-
authority hierarchy. A certificate authority hierarchy is a chain of certificate authorities,
starting with the root-certificate authority, which is the Internet Policy Registration Authority
(IPRA). The IPRA signs certificates using the root key. The root key signs certificates only
for policy-creation authorities, which are organizations that set policies for obtaining digital
certificates. In turn, policy-creation authorities sign digital certificates for CAs. CAs then sign
digital certificates for individuals and organizations. The CA takes responsibility for authen-
tication, so it must check information carefully before issuing a digital certificate. In one case,
human error caused VeriSign to issue two digital certificates to an imposter posing as a
Microsoft employee. Such an error is significant: The inappropriately issued certificates can
cause users to unknowingly download malicious code onto their machines.
Periodically, changing key pairs is necessary to maintain a secure system, as a private key may be compromised without a user’s knowledge. The longer a key pair is used, the more vulnerable the keys are to attack and cryptanalysis. As a result, digital certificates contain an expiration date to force users to switch key pairs. If a private key is compromised before its expiration date, the digital certificate can be canceled, and the user can get a new key pair and digital certificate. Canceled and revoked certificates are placed on a certificate revocation list (CRL). CRLs are stored with the certificate authority that issued the certificates. It is essential for users to report immediately if they suspect that their private keys have been compromised, as the issue of non-repudiation makes certificate owners responsible for anything appearing with their digital signatures. In states with laws dealing with digital signatures, certificates legally bind certificate owners to any transactions involving their certificates.

One problem with CRLs is that they are similar to old paper lists of revoked credit card numbers that were used at the points of sale in stores. This makes for a great inconvenience when checking the validity of a certificate. An alternative to CRLs is the Online Cer-
tificate Status Protocol (OCSP), which validates certificates in real-time. OCSP technology is currently under development. For an overview of OCSP, read “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP” located at ftp.isi.edu/in-notes/rfc2560.txt.

Many people still consider e-commerce unsecure. However, transactions using PKI and digital certificates can be more secure than exchanging private information over phone lines or through the mail. They are even more secure than paying by credit card in person! After all, when you go to a restaurant and the waiter takes your credit card in back to process your bill, how do you know the waiter did not write down your credit-card information? In contrast, the key algorithms used in most secure online transactions are nearly impossible to compromise. By some estimates, the key algorithms used in public-key cryptography are so secure that even millions of today’s computers working in parallel could not break the codes in a century. However, as computing increases, key algorithms that are considered strong today could be broken in the future.

Digital-certificate capabilities are built into many e-mail packages. For example, in Microsoft Outlook, you can go to the Tools menu and select Options. Then click on the Security tab. At the bottom of the dialog box, you will see the option to obtain a digital ID. Selecting the option will take you to a Microsoft Web site with links to several worldwide certificate authorities. Once you have a digital certificate, you can digitally sign your e-mail messages.

To obtain a digital certificate for your personal e-mail messages, visit www.verisign.com or www.thawte.com. VeriSign offers a free 60-day trial, or you can purchase the service for a yearly fee. Thawte offers free digital certificates for personal e-mail. Web server certificates may also be purchased through VeriSign and Thawte; however, they are more expensive than e-mail certificates.

7.10.1 Java Keystores and keytool

Java provides the keytool utility for managing and generating keys, certificates and digital signatures. The keytool utility enables users to generate and import keys in a keystore; you can use the stored keys for functions such as identification verification, encryption and decryption. A keystore is a repository for storing public and private keys. Modifying the set of keys that a keystore contains requires entering that keystore’s password. Note that if no keystore exists, the keytool will create one; the password is set when creating the keystore. The default keystore is in the user’s home directory (e.g., /home/user/.keystore). The -genkey command-line argument produces a public and private key pair and stores that key pair in the keystore. The user can export a certificate based on that key pair using the -export command-line option. To import a trusted certificate from a CA, the -import command is used. To list all of the contents of the keystore, use the -list command. For example, to create a public and private key pair, enter the following at a command prompt:

    keytool -genkey -alias MyCertificate

MyCertificate is an alias for the public and private key pair. The alias simply identifies a particular public and private key pair for later use. Keystores contain aliases for each public and private key pair. Certificates generated with keytool allow for identification through commonName (CN), organizationUnit (OU), organizationName (O), locality-
Name (L), stateName (S) and country (C). When executing the above command line, keytool prompts the user for this information. You then can generate a certificate request to obtain a verified digital certificate from a certificate authority, such as Verisign or Thawte, using the command:

```
keytool -certreq -alias MyCertificate -file myRequest.cer
```

This creates a file called `myRequest.cer` that contains your digital certificate. Follow the instructions on the certificate authority’s Web site to submit your certificate request and obtain your verified certificate.

To create a certificate that others may use to validate your signature, use the command

```
keytool -export -alias MyCertificate -file myCertificate.cer
```

This generates an X.509 certificate that other users can import into their trusted keystores—keystores that contain certificates that the user knows to be correct, such as those from certificate authorities—to validate information with the signature. For more information on the keytool utility, please refer to the documentation at java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html or java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html.

### 7.11 Java Policy Files

The basis of Java security is the Java Sandbox—the protected environment in which Java applications and applets run. This model is similar to placing a child in a sandbox to play; it is a safe environment where certain objects are placed out of reach and can be used only with permission. On a computer, the user must grant an application or applet specific permissions to access certain system resources outside the sandbox. The Java Sandbox security model is comprised of three individual security checks: the bytecode verifier, the class loader and security manager.

If a developer would like to allow certain operations that the security manager would deem potentially dangerous, permissions may be granted on the basis of security policy files. Permissions are comprised of varying levels of access to specific resources. Reading and writing to a file or directory or connecting to an identified port on a host’s machine are two common permissions granted by a policy file.

Permissions may be granted on the basis of the code signer (using signedBy) and the source of the code (using codeBase). Any permission that is not declared explicitly in the policy file is not granted; therefore, it is necessary to have a policy file with at least some content for the JVM to run any applet. All of the parameters for permission granting are defined in the security policy files, which enable the Java Virtual Machine to offer a great level of access control. The security policy files are external text files with certain syntax and class names; note that, as an alternative to learning this syntax, tools such as policytool are available for use with JDK 1.3. Figure 7.9 describes a few of the permissions available for the Java 2 security model.

A system-wide security policy file is responsible for granting code the permission to access files and ports on the entire system. The virtual machine loads this policy file as part of the virtual machine’s initialization. The system-wide policy file (`java.policy`) is in the `lib/security` directory of the Java Runtime Environment (e.g., `C:\Program Files\JavaSoft\JRE\1.3.1\lib\security`). Particular applications can
specify custom security policy files on the command line. Loading other security policy files does not compromise the original system-wide configuration, any modifications are made in addition to the current policy files in use.\textsuperscript{14}

Figure 7.10 presents class \texttt{AuthorizedFileWriter}, which accepts a file path and file body from the command line. Using a \texttt{SecurityManager} to protect against unauthorized access (line 16), lines 26–39 write the specified \texttt{fileBody} to \texttt{file}.

### Permissions

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{java.security.AllPermission}</td>
<td>Grants all possible permissions. Developers should use this permission only for testing purposes as this permission disables all security checks.</td>
</tr>
<tr>
<td>\texttt{java.io.FilePermission}</td>
<td>Grants access to particular sets of files for reading, writing and deleting those files.</td>
</tr>
<tr>
<td>\texttt{java.lang.RuntimePermission}</td>
<td>Grants permissions for modifying runtime behavior, such as the allowing a program to exit the virtual machine, change the source of \texttt{System.in} and queue print jobs.</td>
</tr>
<tr>
<td>\texttt{java.net.SocketPermission}</td>
<td>Grants permission to create socket connections for connecting to other computers over the network. This permission allows fine-grained control over particular ports, host names and connection types.</td>
</tr>
<tr>
<td>\texttt{java.net.NetPermission}</td>
<td>Grants permission to modify to network properties, such as the host with which to validate usernames and passwords.</td>
</tr>
</tbody>
</table>

\textbf{Fig. 7.9} Some permissions available in the Java 2 security model.

1 // AuthorizedFileWriter.java  
2 // AuthorizedFileWriter writes to file using a security manager.  
3 // Permissions must be given via policy files.  
4 package com.deitel.advjhtp1.security.policyfile;  
5  
6 // Java core package  
7 import java.io.*;  
8  
9 public class AuthorizedFileWriter {  
10  
11 // launch application  
12 public static void main( String[] args )  
13 {  
14 // create and set security manager  
15 System.setSecurityManager( new SecurityManager() );
Policy file `authorized.policy` (Fig. 7.11) grants write `FilePermission` for file `authorized.txt`. Should the command line specify a different file, the `SecurityManager` will deny permission to write to it. The following command executes the `AuthorizedFileWriter` application with the `authorized.policy` policy file:

```
java -Djava.security.policy=authorized.policy com.deitel.advjhtp1.security.policyfile.AuthorizedFileWriter "authorized.txt"
```

Policy file `codebase_authorised.policy` (Fig. 7.12) grants the `C:/myclasses` codebase write `FilePermission` for file `codebase_authorised.txt`. If the code is executing from a different codebase, or the command line specifies a different file, the `SecurityManager` will deny permission to write to that file. The following executes the `AuthorizedFileWriter` application with the `codebase_authorised.policy` policy file:
7.12 Digital Signatures for Java Code

Java applets run under strict security restrictions due to the unreliability of code downloaded over public networks. Unlike Java applications, Java applets run in the sandbox by default—an applet developer need not specify a security manager for an applet. Developers who wish to distribute applets with special permissions (e.g., the ability to read or write files on the user’s computer) must sign those applets with digital signatures. This enables users to verify that a signed applet came from a particular company. If the user trusts that company, the user can grant that applet special permissions.

The applet of Fig. 7.13 uses class `FileTreePanel` (similar to class `FileTreeFrame` in Chapter 3) to display a tree of files on the user’s hard drive. The Java sandbox does not allow applets to read or write files on the user’s hard drive, so we must sign the `FileTreeApplet` with a digital signature. When the user’s Web browser downloads the applet and runs it in the Java Plug-in, the plug-in prompts the user with the digital signature of the applet and allows the user to grant permission to the applet.

Signing an applet with a digital signature requires that the signing party stores the applet and its supporting classes in a JAR file. Figure 7.14 lists the contents of the JAR file.

```java
// authorized.policy
// Policy file that grants file write permission
// only to file "authorized.txt"

grant {
  permission java.io.FilePermission
  "authorized.txt", "write";
};

Fig. 7.11 Policy file grants permission to write to file authorized.txt.

// codebase_authorized.policy
// Policy file that grants write permission to
// file "codebase_authorized.txt" for codebase "C:/myclasses"

grant codebase "file:/C:/myclasses" {
  permission java.io.FilePermission
  "codebase_authorized.txt", "write";
};

Fig. 7.12 Policy file grants permission to the specified codebase.

For more information on current and upcoming uses of policy files and permissions in Java, visit the Web sites java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html and java.sun.com/j2se/1.3/docs/guide/security/permissions.html.

Java applets run under strict security restrictions due to the unreliability of code downloaded over public networks. Unlike Java applications, Java applets run in the sandbox by default—an applet developer need not specify a security manager for an applet. Developers who wish to distribute applets with special permissions (e.g., the ability to read or write files on the user’s computer) must sign those applets with digital signatures. This enables users to verify that a signed applet came from a particular company. If the user trusts that company, the user can grant that applet special permissions.

The applet of Fig. 7.13 uses class `FileTreePanel` (similar to class `FileTreeFrame` in Chapter 3) to display a tree of files on the user’s hard drive. The Java sandbox does not allow applets to read or write files on the user’s hard drive, so we must sign the `FileTreeApplet` with a digital signature. When the user’s Web browser downloads the applet and runs it in the Java Plug-in, the plug-in prompts the user with the digital signature of the applet and allows the user to grant permission to the applet.

Signing an applet with a digital signature requires that the signing party stores the applet and its supporting classes in a JAR file. Figure 7.14 lists the contents of the JAR file.
The `keytool` utility enables developers to generate public and private key pairs suitable for signing applets. The Java Plug-in supports applets signed with RSA digital signatures. To generate an RSA key pair, type the following at a command prompt:

```
keytool -genkey -keyalg RSA -alias MyCertificate
```
The `keytool` utility prompts you for your name, organization name and location. To export your digital signature into a certificate file, use the command:

```
keytool -export -alias MyCertificate -file myCertificate.cer
```

The Java Plug-in maintains a keystore for trusted certificates in its `lib/security` folder (e.g., `C:\Program Files\JavaSoft\JRE\1.3.1\lib\security`) named `cacerts`. This keystore contains certificates from certificate authorities such as Verisign and Thawte. The plug-in uses these certificates to verify digital signatures through a certificate chain. Adding a new certificate to this keystore allows the Java Plug-in to verify applets signed with that new certificate. (If your certificate has been signed by a Certificate Authority such as Verisign or Thawte, you need not add this certificate to the `cacerts` trusted keystore.) Add `myCertificate.cer` to the `cacerts` keystore using the command

```
keytool -import -alias MyTrustedCertificate -keystore cacerts -file myCertificate.cer
```

where `cacerts` is the complete path to the `cacerts` keystore in the Java Plug-in’s `lib/security` folder. When prompted for a password, enter `changeit`, which is the `cacert` keystore’s default password.

Next, sign the applet’s JAR file with your digital signature using the `jarsigner` utility. The `jarsigner` utility updates the manifest file in the JAR with the appropriate security information and signs each class in the JAR file. To sign the JAR, enter the following at a command prompt:

```
jarsigner FileTreeApplet.jar MyCertificate
```

Next, create an HTML file that contains an `applet` element for the `FileTreeApplet`. Figure 7.15 contains a basic HTML file for this purpose.

```
<html>
<head>
<title>FileTreeApplet Signed Applet</title>
</head>
<body>
<h1>File Browser</h1>
<applet
code = "com.deitel.advjhtp1.security.signatures.FileTreeApplet"
archive = "FileTreeApplet.jar" width = "400" height = "200">
</applet>
</body>
</html>
```

Fig. 7.15 HTML file for `FileTreeApplet`. 

To enable the Web browser to load the Java Plug-in instead of the Web browser’s own Java Virtual Machine, use the `htmlconverter` utility to convert the `applet` element into appropriate object and embed elements using the command

```
htmlconverter signedApplet.html
```

and load the resulting Web page in a Web browser. When the Java Plug-in loads the applet, the plug-in displays the `Java Plug-in Security Warning` dialog (Fig. 7.16). This dialog displays information about the signing certificate and enables the user to grant special permission to the applet. The user can click **Grant this Session** to allow the applet **AllPermission** for the current browsing session, **Deny** to deny special permission to the applet, **Grant Always** to allow the applet **AllPermission** for this and future browsing sessions, or **More Info** to display detailed information about the applet’s signature.

Figure 7.17 shows the applet running with **AllPermission**. The applet prompts the user to enter a directory to use as the root of the `JTree`. The user then can browse through the filesystem and click on individual files or folders to view information about those files and folders in the right-hand pane. The user also can rename files in the `JTree`.

---

**Fig. 7.16**  Java Plug-in security warning when loading a signed applet.

**Fig. 7.17**  `FileTreeApplet` browsing the `D:\jdk1.3.1\` directory.
7.13 Authentication

Ensuring that users actually are who they claim to be is a large part of computer security, known as authentication. Current authentication models restrict access to certain aspects of a program, allow users to connect to a network and regulate the resources available to users on the network. Java uses the Java Authentication and Authorization Service (JAAS) for authenticating and authorizing users. JAAS is based on a plug-in framework, which allows Kerberos and single sign-on to be implemented for authentication and authorization.

7.13.1 Kerberos

Kerberos is a freely available open-source protocol developed at MIT. It employs secret-key cryptography to authenticate users in a network and to maintain the integrity and privacy of network communications.

Authentication in a Kerberos system is handled by the main Kerberos system and a secondary Ticket Granting Service (TGS). The latter system is similar to key distribution centers, which were described in Section 7.3. The main Kerberos system authenticates a client’s identity to the TGS, which in turn authenticates client’s rights to access specific network services.

Each client in the network shares a secret key with the Kerberos system. This secret key may be used by multiple TGSs in the Kerberos system. The client starts by entering a login name and password into the Kerberos authentication server, which maintains a database of all clients in the network. The authentication server returns a Ticket-Granting Ticket (TGT) encrypted with the client’s secret key that it shared with the authentication server. Since the secret key is known only by the authentication server and the client, only the client can decrypt the TGT, thus authenticating the client’s identity. Next, the client sends the decrypted TGT to the Ticket Granting Service to request a service ticket, which authorizes the client’s access to specific network services. Service tickets have a set expiration time. Tickets may be renewed by the TGS.

7.13.2 Single Sign-On

To access multiple applications on different servers, users must provide a separate password for authentication on each. Remembering multiple passwords is cumbersome. People tend to write their passwords down, creating security threats.

Single sign-on systems allow users to log in once with a single password. Users can access multiple applications. It is important to secure single sign-on passwords, because if the password becomes available to hackers, all applications can be accessed and attacked.

There are three types of single sign-on services: workstation logon scripts, authentication server scripts and tokens. Workstation logon scripts are the simplest form of single sign-on. Users log in at their workstations, then choose applications from a menu. The workstation logon script sends the user’s password to the application servers, and the user is authenticated for future access to those applications. Workstation logon scripts do not provide a sufficient amount of security since user passwords are stored on the workstation in plaintext. Anyone who can access the workstation can obtain the user’s password. Authentication server scripts authenticate users with a central server. The central server controls connections between the user and the applications the user wishes to access. Authentication server scripts are more secure than workstation logon scripts because passwords are kept on the server, which is more secure than the individual PC.
The most advanced single sign-on systems use token-based authentication. Once a user is authenticated, a non-reusable token is issued to the user to access specific applications. The logon for creating the token is secured with encryption or with a single password, which is the only password the user needs to remember or change. The only problem with token authentication is that all applications must be built to accept tokens instead of traditional logon passwords.15

7.13.3 Java Authentication and Authorization Service (JAAS)

Java addresses the problems often associated with authenticating users and controlling access with the Java Authentication and Authorization Service (JAAS) API. Whereas policy files and permissions protect a user from running malicious programs, JAAS protects applications from unauthorized users.16

The Pluggable Authentication Module (PAM) architecture is the standard method for authentication on which JAAS is based.17 The PAM framework supports multiple authentication systems, including Kerberos tickets and smart cards. Additionally, PAM allows different systems to be combined to create even greater levels of security. Developers determine what forms of authentication will be used in the associated security policy. PAM also supports single sign-on systems. Java’s implementation of PAM in JAAS enables Java programs to identify users, allowing developers to establish access controls to protect those programs from unauthorized access.

After a user has been authenticated, JAAS can grant or restrict access to certain resources of an application. JAAS can control access by group, user or role-based security policies.18 User-based access control governs access to resources on an individual user basis. After providing a password, Kerberos ticket or other means of identification to the Java application, the privileges of the individual user are determined and applied. Group-based authorization identifies a user as a part of a group and grants access to certain resources based on the identifying group. For example, a member of the group “doctors” would be able to access patient databases, connect to remote hospitals and write prescriptions that are sent electronically to a pharmacy. Role-based access control (RBAC) is used in addition to group-access control, allowing for more control over resources. Users request specific roles, each of which have corresponding privileges, based on what tasks the user would need to access. Roles and the corresponding permissions are based on the makeup of an organization. What separates roles from groups is the fact that by default, roles are not enabled. This feature increases security by allowing users to access only necessary applications. Using the “doctors” example, it would be risky for an application to allow the doctor to delete a patient’s profile by default. Deleting patient files should be an available option only when it is necessary. Therefore, in order to gain access to a deletion resource, the doctor would have to present additional identification.19

Class AuthenticateNT (Fig. 7.18) uses a sample login module available for JAAS that authenticates the current user with the Windows NT authentication system. Lines 20–21 create a new LoginContext with the name AuthenticateNT. This LoginContext is associated with a specific login module in the configuration file of Fig. 7.20. Line 24 invokes method login of class LoginContext. This begins the authentication process. The Windows NT sample login module does not prompt the user for login information, it simply obtains the credentials for the currently logged-in user. Other login modules may use a CallbackHandler to prompt the user to enter a username, password and
other authentication information. If the login is successful (i.e., invoking method \texttt{login} does not generate any exceptions), line 28 prints a message indicating so.

```
// AuthenticateNT.java
// Authenticates a user using the NTLoginModule and performs
// a WriteFileAction PrivilegedAction.
package com.deitel.advjhtp1.security.jaas;

// Java extension packages
import javax.swing.*;
import javax.security.auth.*;
import javax.security.auth.login.*;

public class AuthenticateNT {

    // launch application
    public static void main( String[] args )
    {
        // authenticate user and perform PrivilegedAction
        try {

            // create LoginContext for AuthenticateNT context
            LoginContext loginContext =
                new LoginContext( "AuthenticateNT" );

            // perform login
            loginContext.login();

            // if login executes without exceptions, login
            // was successful
            System.out.println( "Login Successful" );

            // get Subject now associated with LoginContext
            Subject subject = loginContext.getSubject();

            // display Subject details
            System.out.println( subject );

            // perform the WriteFileAction as current Subject
            Subject.doAs( subject, new WriteFileAction() );

            // log out current Subject
            loginContext.logout();

            System.exit( 0 );
        } // end try

        // handle exception loggin in
        catch ( LoginException loginException ) {
            loginException.printStackTrace();
        }
    }
}
```

\textit{Fig. 7.18} \texttt{AuthenticateNT} uses the \texttt{NTLoginModule} to authenticate a user and invoke a \texttt{PrivilegedAction} (part 1 of 2).
Line 31 obtains a **Subject** from the current **LoginContext**. A **Subject** represents a particular user or other entity (e.g., an automated service) that requests an action. Each **Subject** has associated **Principals**. These **Principals** represent the different roles or identities that a user can assume during a particular login session. The security

![Login Successful](image)

**Fig. 7.18**  **AuthenticateNT** uses the **NTLoginModule** to authenticate a user and invoke a **PrivilegedAction** (part 2 of 2).
restrictions in place for a particular application can grant permissions for Principals to make certain requests (e.g., read from a particular file). Line 34 prints the Subject’s information, including a list of the Subject’s Principals.

Line 37 invokes method doAs of class Subject to make a request using the given Subject. Method doAs takes as arguments the Subject for the request and a PrivilegedAction that contains the request. For this example, line 37 passes a new WriteFileAction (Fig. 7.19), which writes a simple message to a text file. Line 40 logs out from the current LoginContext.

Class WriteFileAction (Fig. 7.19) implements interface PrivilegedAction. PrivilegedActions execute in the context of an AccessController, which verifies that the Subject invoking the PrivilegedAction has the appropriate permissions. Interface PrivilegedAction requires that implementations define method run (lines 13–32). Method run of class WriteFileAction creates a text file and writes a message to that text file.

```
// WriteFileAction.java
// WriteFileAction is a PrivilegedAction implementation that
// simply writes a file to the local file system.
package com.deitel.advjhtp1.security.jaas;

// Java core packages
import java.io.*;
import java.security.PrivilegedAction;

public class WriteFileAction implements PrivilegedAction {

    // perform the PrivilegedAction
    public Object run() {
        // attempt to write a message to the specified file
        try {
            File file = new File( "D:/", "privilegedFile.txt" );
            FileWriter fileWriter = new FileWriter( file );

            // write message to File and close FileWriter
            fileWriter.write( "Welcome to JAAS!" );
            fileWriter.close();
        }

        // handle exception writing file
        catch ( IOException ioException ) {
            ioException.printStackTrace();
        }

        return null;
    }
}
```

Fig. 7.19 WriteFileAction is a PrivilegedAction for writing a simple text file.
The configuration file of Fig. 7.20 specifies the `LoginModules` to use for the `AuthenticateNT LoginContext`. Line 5 specifies that the `Subject` must authenticate with the `NTLoginModule` for a successful login. LoginContexts can require a sequence of several `LoginModules` for proper authentication. For more information on JAAS configuration files, please refer to the JAAS documentation at `java.sun.com/security/jaas/doc/api.html`.

The policy file of Fig. 7.21 grants permissions to the specified `Principal` when executing code in the specified `codeBase`. JAAS offers fine-grained permissions control. This example grants read and write `FilePermission` to the `Principal "santry"` when executing code in the `file:d:/JavaProjects/advjhtpl/src/-` codebase.

The policy file of Fig. 7.22 specifies permissions for JAAS itself and for the `AuthenticateNT` class codebase. Lines 5–7 grant `AllPermission` to the JAAS standard extension. This permission enables JAAS to perform authentication on behalf of this application. Line 13 grants permission to execute `PrivilegedAction`s using method `doAs`. Lines 15–19 grant permission to read and write the text file `D:\privileged-File.txt`.

---

```java
// jaas.config
// Configures JAAS to use NTLoginModule
// for authentication.
AuthenticateNT {
    com.sun.security.auth.module.NTLoginModule required debug=false;
};
```

**Fig. 7.20** Configuration file for authentication using `NTLoginModule`.

```java
// jaas.policy
// Policy file defining the permissions for the named Principal
grant codeBase "file:D:/JavaProjects/advjhtpl/src/-",
    Principal com.sun.security.auth.NTUserPrincipal "santry" {
    permission java.io.FilePermission "D:/privilegedFile.txt",
        "write";
    permission java.io.FilePermission "D:/privilegedFile.txt",
        "read";
};
```

**Fig. 7.21** JAAS policy file for granting permissions to a `Principal` and codebase.

```java
// java.policy
// Policy file that grants AllPermission
// to JAAS modules and specific permissions
// to the D:\Projects\Java codebase.
grant codebase "file:D:/jdk1.3.1/jre/lib/ext/jaas.jar" {
    permission java.security.AllPermission;
};
```

**Fig. 7.22** Policy file for JAAS application (part 1 of 2).
Executing the **AuthenticateNT** example requires several command-line options to the Java virtual machine. Enter the following at a command prompt:

```java
java -Djava.security.policy=java.policy
   -Djava.security.auth.policy=jaas.policy
   -Djava.security.auth.login.config=jaas.config
   com.deitel.advjhtp1.security.jaas.AuthenticateNT
```

where **java.policy** is the policy file of Fig. 7.22, **jaas.policy** is the policy file of Fig. 7.21 and **jaas.config** is the configuration file of Fig. 7.20.

### 7.14 Secure Sockets Layer (SSL)

Currently, most e-businesses use SSL for secure online transactions, although SSL is not designed specifically for securing transactions. Rather, SSL secures World Wide Web connections. The Secure Sockets Layer (SSL) protocol, developed by Netscape Communications, is a nonproprietary protocol commonly used to secure communication between two computers on the Internet and the Web.\(^{20}\) SSL is built into many Web browsers, including Netscape **Communicator** and Microsoft **Internet Explorer**, as well as numerous other software products. It operates between the Internet’s TCP/IP communications protocol and the application software.\(^{21}\)

SSL implements public-key technology using the RSA algorithm and digital certificates to authenticate the server in a transaction and to protect private information as it passes from one party to another over the Internet. SSL transactions do not require client authentication; many servers consider a valid credit-card number to be sufficient for authentication in secure purchases. To begin, a client sends a message to a server. The server responds and sends its digital certificate to the client for authentication. Using public-key cryptography to communicate securely, the client and server negotiate *session keys* to continue the transaction. Session keys are secret keys used for the duration of that transaction. Once the keys are established, the communication proceeds between the client and the server using the session keys and digital certificates. Encrypted data are passed through TCP/IP, just as regular packets travel over the Internet. However, before sending a message with TCP/IP, the SSL protocol breaks the information into blocks and compresses and encrypts those blocks. Conversely, after the data reach the receiver through
TCP/IP, the SSL protocol decrypts the packets, then decompresses and assembles the data. These extra processes provide an extra layer of security between TCP/IP and applications. SSL is used primarily to secure point-to-point connections—transmissions of data from one computer to another.\textsuperscript{22} SSL allows for the authentication of the server, the client, both or neither; in most Internet SSL sessions, only the server is authenticated. The Transport Layer Security (TLS) protocol, designed by the Internet Engineering Task Force, is similar to SSL. For more information on TLS, visit: \url{www.ietf.org/rfc/rfc2246.txt}.

Although SSL protects information as it is passed over the Internet, it does not protect private information, such as credit-card numbers, once the information is stored on the merchant’s server. When a merchant receives credit-card information with an order, the information is often decrypted and stored on the merchant’s server until the order is placed. If the server is not secure and the data are not encrypted, an unauthorized party can access the information. For more information about the SSL protocol, check out the Netscape SSL tutorial at \url{developer.netscape.com/tech/security/ssl/protocol.html} and the Netscape Security Center site at \url{www.netscape.com/security/index.html}.

7.14.1 Java Secure Socket Extension (JSSE)

The strength of SSL encryption has been integrated into Java technology through Sun’s \textit{Java Secure Socket Extension (JSSE)}. Java applications that use JSSE can secure a passage between a client and a server over TCP/IP. JSSE provides encryption, message integrity checks and authentication of the server and client.\textsuperscript{23} JSSE uses keystores to secure storage of key pairs and certificates used in PKI. A \textit{truststore} is a keystore containing keys and certificates used to validate the identities of servers and clients.\textsuperscript{24, 25}

The algorithms used in JSSE for encryption, key agreement and authentication include DES, 3DES, Diffie-Hellman and DSA. Like JCE, JSSE uses a provider-based model that enables third parties to provide additional cryptographic algorithms. JSSE is free for commercial use and is available for free download at \url{java.sun.com/products/jsse}.

Class \texttt{LoginServer} (Fig. 7.23) uses an \texttt{SSLServerSocket} to listen for SSL connections on port \texttt{7070}. JSSE uses the Factory design pattern for constructing \texttt{SSLServerSockets} and \texttt{SSLSockets}. Line 25 invokes \texttt{static} method \texttt{getDefault} of class \texttt{SSLServerSocketFactory} to obtain the default \texttt{SSLServerSocketFactory}. Line 29 invokes method \texttt{createServerSocket} of class \texttt{SSLServerSocketFactory} to create the \texttt{SSLSocket}. This method takes as an argument the port number on which the \texttt{SSLSocket} will listen.

Method \texttt{runServer} (lines 34–84) starts \texttt{LoginServer}. Line 46 invokes method \texttt{accept} of class \texttt{SSLSocket} to accept a new client connection. Method \texttt{accept} is a blocking call that returns an \texttt{SSLSocket} when a client connects. Lines 49–55 obtain the \texttt{InputStream} and \texttt{OutputStream} for the \texttt{SSLSocket} and lines 57–58 read two lines of text. Lines 60–61 validate the client’s user name and password against constants \texttt{CORRECT_USER_NAME} and \texttt{CORRECT_PASSWORD}. If the user name and password are correct, line 63 sends a welcome message to the client. If the user name and password are incorrect, line 67 notifies the client that the login failed. Lines 71–73 close the \texttt{InputStream}, \texttt{OutputStream} and \texttt{SSLSocket}.

Note that using \texttt{SSLServerSockets} and \texttt{SSLSockets} is identical to using standard \texttt{ServerSockets} and \texttt{Sockets}. JSSE hides the details of the SSL protocol and encryption from the programmer entirely.
LoginServer.java
LoginServer uses an SSLServerSocket to demonstrate JSSE's SSL implementation.

```java
package com.deitel.advjhtp1.security.jsse;

// Java core packages
import java.io.*;

// Java extension packages
import javax.net.ssl.*;

public class LoginServer {

    private static final String CORRECT_USER_NAME = "Java";
    private static final String CORRECT_PASSWORD = "HowToProgram";

    private SSLServerSocket serverSocket;

    // LoginServer constructor
    public LoginServer() throws Exception {
        // SSLServerSocketFactory for building SSLServerSockets
        SSLServerSocketFactory socketFactory = (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();

        // create SSLServerSocket on specified port
        serverSocket = (SSLServerSocket)socketFactory.createServerSocket(7070);
    }

    // start server and listen for clients
    private void runServer() {
        // perpetually listen for clients
        while (true) {
            // wait for client connection and check login information
            try {
                System.err.println("Waiting for connection...");

                // create new SSLSocket for client
                SSLSocket socket = (SSLSocket)serverSocket.accept();

                // open BufferedReader for reading data from client
                BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            }
        }
    }
}
```

Fig. 7.23 LoginServer uses an SSLServerSocket for secure communication (part 1 of 2).
Class LoginClient (Fig. 7.24) uses an SSLSocket to communicate with the LoginServer. Lines 22–23 invoke method getDefault of class SSLSocketFactory to obtain the default SSLSocketFactory. Lines 26–28 invoke method createSocket of class SSLSocketFactory to create a new SSLSocket that connects to localhost on port 7070. Lines 31–32 create a new PrintWriter for the SSLSocket's OutputStream to facilitate sending data to the server. Lines 35–48 prompt the user for a username and password and send them to the server. Lines 51–58 then read the response from the server and display this response in a JOptionPane message.
dialog. Note that once the client establishes the connection with the `SSLSocket`, the fact that SSL encrypts the communication between the client and server is transparent to the programmer.

```java
// LoginClient.java
// LoginClient uses an SSLSocket to transmit fake login
// information to LoginServer.
package com.deitel.advjhtp1.security.jsse;

// Java core packages
import java.io.*;

// Java extension packages
import javax.swing.*;
import javax.net.ssl.*;

public class LoginClient {

    // LoginClient constructor
    public LoginClient()
    {
        // open SSLSocket connection to server and send login
        try {
            // obtain SSLSocketFactory for creating SSLSockets
            SSLSocketFactory socketFactory =
                ( SSLSocketFactory ) SSLSocketFactory.getDefault();

            // create SSLSocket from factory
            SSLSocket socket =
                ( SSLSocket ) socketFactory.createSocket(
                    "localhost", 7070 );

            // create PrintWriter for sending login to server
            PrintWriter output = new PrintWriter(
                new OutputStreamWriter( socket.getOutputStream() ) );

            // prompt user for user name
            String userName = JOptionPane.showInputDialog( null,
                "Enter User Name:" );

            // send user name to server
            output.println( userName );

            // prompt user for password
            String password = JOptionPane.showInputDialog( null,
                "Enter Password:" );

            // send password to server
            output.println( password );

            output.flush();
        }
    }
}
```

**Fig. 7.24** *LoginClient* communicates with *LoginServer* via SSL (part 1 of 2).
Enabling SSL requires that LoginServer uses a certificate that LoginClient trusts. Use the keytool to generate a new certificate and keystore for this purpose:

```
keytool -genkey -keystore SSLStore -alias SSLCertificate
```

Next, launch LoginServer and specify the keystore that contains the LoginServer’s certificate:

```
java -Djavax.net.ssl.keyStore=SSLStore
    -Djavax.net.ssl.keyStorePassword=password
com.deitel.advjhtp1.security.jsse.LoginServer
```

where password is the password you specified for the SSLStore keystore. Finally, launch the LoginClient and specify the truststore for that client. The truststore contains certificates that the client trusts for the purposes of digital-signature validation. For simplicity in this example, we use the same keystore as both the LoginServer’s keystore and the Login-
Client’s truststore. In real-world applications, the client’s truststore should contain trusted certificates, such as those from certificate authorities. Execute the client using the command

```
java -Djavax.net.ssl.trustStore=SSLStore
     -Djavax.net.ssl.trustStorePassword=password
     com.deitel.advjhtp1.security.jsse.LoginClient
```

Figure 7.25 shows two executions of class LoginClient. The first execution (the left column) shows a successful login. The second execution shows a failed login. For each execution, the LoginServer and LoginClient used SSL to encrypt all data transfer.

### 7.15 Java Language Security and Secure Coding

The Java language provides programmers with a security advantage that other languages do not. Java code goes through several stages before and during execution that help to ensure that the Java code is not malicious. Each stage prevents many exploitations that other programming languages allow. For example, the Java Virtual Machine ensures that programs cannot read memory beyond the end of an array. This prevents programs from reading data from arbitrary locations in memory.

The Java compiler performs several security checks in the normal process of compiling Java source code into bytecode. The compiler ensures that the program does not read from uninitialized variables—a common technique for reading data from arbitrary memory locations. Also, the compiler checks the access modifiers for each method invocation and variable to ensure that the program accesses private data only from the proper classes. The compiler also can detect certain illegal casts between data types. These simple steps help prevent many types of security attacks.

---

Fig. 7.25 Two sample executions of class LoginClient.
Before the Java Virtual Machine executes a Java program, the bytecodes for that program must pass through the byte code verifier. The bytecode verifier ensures that the bytecodes are valid Java bytecodes that do not perform certain illegal operations (according to the rules of the Java language). For example, the bytecode verifier checks that no class has more than one superclass and that final classes do not have subclasses. The bytecode verifier also checks that classes have the proper format. This second stage helps eliminate security risks that the compiler could not detect.

The Java Virtual Machine performs the remaining integrity checks on Java programs. The virtual machine checks any remaining cast operations to ensure their validity and performs array-bounds checking to prevent programs from reading arbitrary memory locations. The Java Virtual Machine uses class loaders to read class definitions from .class files and produce class representations in memory. The class loader uses separate namespaces to prevent malicious code from interacting with safe code. Every class that the JVM loads is assigned a corresponding set of unique names, known as a namespace. Namespaces in Java act as barriers between classes. The JVM allows classes within the same namespace to interact, but requires explicit permission for classes from separate namespaces to operate together. To protect trusted classes, developers can use different class loaders for trusted and untrusted packages. This allows for trusted code to belong to the same runtime package. A runtime package is code that was loaded by the same class loader and belong to the same package. The bootstrap class loader loads all of the trusted core Java classes, so that additional, potentially unsecure classes will belong to a separate runtime package and can not interfere with the core classes.

The Java security manager is the module that performs security checks while code is running. If, during run-time, code attempts to execute a dangerous operation, a security exception is generated. Class operations that the security manager considers dangerous include: deleting a file, reading from a file, appending or editing a file, adding or loading a class to a package, and opening a socket connection. When untrusted code attempts one of these operations, the security manager throws an AccessControlException.

Although the Java language and the associated compiler, bytecode verifier and virtual machine enforce a number of security constraints, Java developers still may need to take certain steps to secure their code. For example, when developing classes, programmers can declare those classes as final. Final classes cannot be subclassed, and can help make an application more secure by preventing attackers from creating malicious subclasses that can cause damage to an application or gain access to protected information. A hacker would generally exploit an ordinary class by creating a subclass to replace the original. This new class could expose the same public interface as the original class, and would deceive the compiler into using the hostile class.

Fortunately, the compiler will not compile a class that attempts to extent a final class. The bytecode verifier also checks to ensure that classes are not subclasses of a final class. If declaring an entire class as final is excessive for a program’s needs, final methods prevent subclasses from overriding particular methods.

7.16 Internet and World Wide Web Resources

The notation <www.domain-name.com> indicates that the citation is for information found at that Web site.
**Security Resource Sites**

**www.securitysearch.com**
This is a comprehensive resource for computer security. The site has thousands of links to products, security companies, tools and more. The site also offers a free weekly newsletter with information about vulnerabilities.

**www.securityfocus.com**
A site covering all aspects of computer security, with special sections in the basics, numerous operating systems, intrusion detection and viruses. This site is also responsible for maintaining the BugTraq list.

**www.esecurityonline.com**
This site is a great resource for information on online security. The site has links to news, tools, events, training and other valuable security information and resources.

**www.epic.org**
The Electronic Privacy Information Center deals with protecting privacy and civil liberties. Visit this site to learn more about the organization and its latest initiatives.

**theory.lcs.mit.edu/~rivest/crypto-security.html**
The Ronald L. Rivest: Cryptography and Security site has an extensive list of links to security resources, including newsgroups, government agencies, FAQs, tutorials and more.

**www.w3.org/Security/Overview.html**
The W3C Security Resources site has FAQs, information about W3C security and e-commerce initiatives and links to other security related Web sites.

**web.mit.edu/network/ietf/sa**
The Internet Engineering Task Force (IETF), which is an organization concerned with the architecture of the Internet, has working groups dedicated to Internet Security. Visit the IETF Security Area to learn about the working groups, join the mailing list or check out the latest drafts of the IETF’s work.

**dir.yahoo.com/Computers_and_Internet/Security_and_Encryption**
The Yahoo Security and Encryption page is a great resource for links to Web sites security and encryption.

**www.counterpane.com/hotlist.html**
The Counterpane Internet Security, Inc., site includes links to downloads, source code, FAQs, tutorials, alert groups, news and more.

**www.rsasecurity.com/rsalabs/faq**
This site is an excellent set of FAQs about cryptography from RSA Laboratories, one of the leading makers of public key cryptosystems.

**www.nsi.org/compsec.html**
Visit the National Security Institute’s Security Resource Net for the latest security alerts, government standards, and legislation, as well as security FAQs links and other helpful resources.

**www.itaa.org/infosec**
The Information Technology Association of America (ITAA) InfoSec site has information about the latest U.S. government legislation related to information security.

**staff.washington.edu/dittrich/misc/ddos**
The Distributed Denial of Service Attacks site has links to news articles, tools, advisory organizations and even a section on security humor.

**www.infoworld.com/cgi-bin/displayNew.pl?/security/links/security_corner.htm**
The Security Watch site on Infoword.com has loads of links to security resources.
www.antionline.com
AntiOnline has security-related news and information, a tutorial titled “Fight-back! Against Hackers,” information about hackers and an archive of hacked sites.

www.microsoft.com/security/default.asp
The Microsoft security site has links to downloads, security bulletins and tutorials.

www.grc.com
This site offers a service to test the security of your computer’s Internet connection.

General Security Sites

www.sans.org/giac.html
Sans Institute presents information on system and security updates, along with new research and discoveries. The site offers current publications, projects, and weekly digests.

www.packetstorm.securify.com
The Packet Storm page describes the twenty latest advisories, tools, and exploits. This site also provides links to the top security news stories.

www.xforce.iss.net
This site allows one to search a virus by name, reported date, expected risk, or affected platforms. Updated news reports can be found on this page.

www.ntbugtraq.com
This site provides a list and description of various Windows NT Security Exploits/Bugs encountered by Windows NT users. One can download updated service applications.

nsi.org/compsec.html

www.tno.nl/instit/fel/intern/wkinfsec.html
This site includes numerous links to other security sites.

www.microsoft.com/security
The Microsoft security site offers news, product information and tools.

www.securitystats.com
This computer security site provides statistics on viruses, web defacements and security spending.

Magazines, Newsletters and News sites

www.networkcomputing.com/consensus
The Security Alert Consensus is a free weekly newsletter with information about security threats, holes, solutions and more.

www.infosecuritymag.com
Information Security Magazine has the latest Web security news and vendor information.

www.issl.org/cipher.html
Cipher is an electronic newsletter on security and privacy from the Institute of Electrical and Electronics Engineers (IEEE). You can view current and past issues online.

securityportal.com
The Security Portal has news and information about security, cryptography and the latest viruses.

www.scmagazine.com
SC Magazine has news, product reviews and a conference schedule for security events.

www.cnn.com/TECH specials/hackers
Insurgency on the Internet from CNN Interactive has news on hacking, plus a gallery of hacked sites.
Visit Rootshell for security-related news and white papers.

**Government Sites for Computer Security**

**www.cit.nih.gov/security.html**
This site has links to security organizations, security resources and tutorials on PKI, SSL and other protocols.

**cs-www.ncsl.nist.gov**
The *Computer Security Resource Clearing House* is a resource for network administrators and others concerned with security. This site has links to incident-reporting centers, information about security standards, events, publications and other resources.

**www.ctd.org/crypto**
Visit the Center for Democracy and Technology for U. S. legislation and policy news regarding cryptography.

**www.epm.ornl.gov/~dunigan/security.html**
This site has links to loads of security-related sites. The links are organized by subject and include resources on digital signatures, PKI, smart cards, viruses, commercial providers, intrusion detection and several other topics.

**www.alw.nih.gov/Security**
The *Computer Security Information* page is an excellent resource, providing links to news, newsgroups, organizations, software, FAQs and an extensive number of Web links.

**www.fedcirc.gov**
The Federal Computer Incident Response Capability deals with the security of government and civilian agencies. This site has information about incident statistics, advisories, tools, patches and more.

**axion.physics.ubc.ca/pgp.html**
This site has a list of freely available cryptosystems, along with a discussion of each system and links to FAQs and tutorials.

**www.ifccfbi.gov**
The Internet Fraud Complaint Center, founded by the Justice Department and the FBI, fields reports of Internet fraud.

**www.disa.mil/infosec/iaweb/default.html**
The Defense Information Systems Agency’s *Information Assurance* page includes links to sites on vulnerability warnings, virus information and incident-reporting instructions, as well as other helpful links.

**www.nswc.navy.mil/ISSEC/**
The objective of this site is to provide information on protecting your computer systems from security hazards. Contains a page on hoax versus real viruses.

**www.cit.nih.gov/security.html**
You can report security issues at this site. The site also lists official federal security policies, regulations, and guidelines.

**cs-www.ncsl.nist.gov/**
The Computer Security Resource Center provides services for vendors and end users. The site includes information on security testing, management, technology, education and applications.

**Advanced Encryption Standard (AES)**

**csrc.nist.gov/encryption/aes**
This is the official site for the AES; this site includes press releases and a discussion forum.
www.esat.kuleuven.ac.be/~rijmen/rijndael/
Visit this site for information about the Rijndael algorithm, including links to various implementations of the algorithm and a small FAQ.

home.ecn.ab.ca/~jsavard/crypto/co040801.htm
This site is dedicated to AES. It includes an explanation of the algorithm with diagrams and examples.

**Internet Security Vendors**

www.rsasecurity.com
RSA is one of the leaders in electronic security. Visit its site for more information about its current products and tools, which are used by companies worldwide.

www.ca.com/protection
*Computer Associates* is a vendor of Internet security software. It has various software packages to help companies set up a firewall, scan files for viruses and protect against viruses.

www.checkpoint.com
Check Point™ Software Technologies Ltd. is a leading provider of Internet security products and services.

www.opsec.com
The Open Platform for Security (OPSEC) has over 200 partners that develop security products and solutions using the OPSEC to allow for interoperability and increased security over a network.

www.baltimore.com
Baltimore Security is an e-commerce security solutions provider. Their UniCERT digital certificate product is used in PKI applications.

www.ncipher.com
nCipher is a vendor of hardware and software products, including an SSL accelerator that increases the speed of secure Web server transactions and a secure key management system.

www.entrust.com
Entrust Technologies provides e-security products and services.

www.tenfour.co.uk
TenFour provides software for secure e-mail.

www.antivirus.com
*ScanMail®* is an e-mail virus detection program for Microsoft Exchange.

www.contenttechnologies.com/ads
Content Technologies is a security software provider. Its products include firewall and secure e-mail programs.

www.zixmail.com
Zixmail™ is a secure e-mail product that allows you to encrypt and digitally sign your messages using different e-mail programs.

web.mit.edu/network/pgp.html
Visit this site to download *Pretty Good Privacy®* freeware. PGP allows you to send messages and files securely.

www.certicom.com
Certicom provides security solutions for the wireless Internet.

www.raytheon.com
Raytheon Corporation’s *SilentRunner* monitors activity on a network to find internal threats, such as data theft or fraud.
SSL

developer.netscape.com/tech/security/ssl/protocol.html
This Netscape page has a brief description of SSL, plus links to an SSL tutorial and FAQs.

www.netscape.com/security/index.html
The Netscape Security Center is an extensive resource for Internet and Web security. You will find news, tutorials, products and services on this site.

psych.psy.uq.oz.au/~ftp/Crypto
This FAQs page has an extensive list of questions and answers about SSL technology.

www.visa.com/nt/ecomm/security/main.html
Visa International’s security page includes information on SSL and SET. The page includes a demonstration of an online shopping transaction, which explains how SET works.

www.openssl.org
The Open SSL Project provides a free, open source toolkit for SSL.

Public-key Cryptography

www.entrust.com
Entrust produces effective security software products using Public Key Infrastructure (PKI).

www.cse.dnd.ca
The Communication Security Establishment has a short tutorial on Public Key Infrastructure (PKI) that defines PKI, public-key cryptography and digital signatures.

www.magnet.state.ma.us/itd/legal/pki.htm
The Commonwealth of Massachusetts Information Technology page has loads of links to sites related to PKI that contain information about standards, vendors, trade groups and government organizations.

www.ftech.net/~monark/crypto/index.htm
The Beginner’s Guide to Cryptography is an online tutorial and includes links to other sites on privacy and cryptography.

www.faqs.org/faqs/cryptography-faq
The Cryptography FAQ has an extensive list of questions and answers.

www.pkiforum.org
The PKI Forum promotes the use of PKI.

www.counterpane.com/pki-risks.html
Visit the Counterpane Internet Security, Inc.’s site to read the article “Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure.”

Digital Signatures

www.ietf.org/html.charters/xmldsig-charter.html
The XML Digital Signatures site was created by a group working to develop digital signatures using XML. You can view the group’s goals and drafts of their work.

www.elock.com
E-Lock Technologies is a vendor of digital-signature products used in Public Key Infrastructure. This site has an FAQs list covering cryptography, keys, certificates and signatures.

www.digsigtrust.com
The Digital Signature Trust Co. is a vendor of Digital Signature and Public Key Infrastructure products. It has a tutorial titled “Digital Signatures and Public Key Infrastructure (PKI) 101.”
Digital Certificates

www.verisign.com
VeriSign creates digital IDs for individuals, small businesses and large corporations. Check out its Web site for product information, news and downloads.

www.thawte.com
Thawte Digital Certificate Services offers SSL, developer and personal certificates.

www.silanis.com/index.htm
Silanis Technology is a vendor of digital-certificate software.

www.belsign.be
Belsign issues digital certificates in Europe. It is the European authority for digital certificates.

www.certco.com
Certco issues digital certificates to financial institutions.

www.openca.org
Set up your own CA using open-source software from The OpenCA Project.

Kerberos

This site is an extensive list of FAQs on Kerberos from the Naval Research Laboratory.

web.mit.edu/kerberos/www
Kerberos: The Network Authentication Protocol is a list of FAQs provided by MIT.

www.contrib.andrew.cmu.edu/~shadow/kerberos.html
The Kerberos Reference Page has links to several informational sites, technical sites and other helpful resources.

www.pdc.kth.se/kth-krb
Visit this site to download various Kerberos white papers and documentation.

Newsgroups

news://comp.security.firewalls
news://comp.security.unix
news://comp.security.misc
news://comp.protocols.kerberos

SUMMARY

• There are five fundamental requirements of a successful, secure transaction: privacy, integrity, authentication, authorization and nonrepudiation.

• Network security addresses the issue of availability: How do we ensure that the network and the computer systems it connects will stay in operation continuously?

• The Java Sandbox architecture and policy files protect users and systems from malicious programs that would otherwise crash computers or steal valuable information.

• To secure information, data can be encrypted. Cryptography transforms data by using a cipher, or cryptosystem—a mathematical algorithm for encrypting messages. Unencrypted data is called plaintext; encrypted data is called ciphertext.

• A key—a string of alpha-numeric characters that acts as a password—is input to the cipher. The cipher uses the key to make data incomprehensible to all but the sender and intended receivers.
• Encryption and decryption keys are binary strings with a given key length. Symmetric cryptography, also known as secret-key cryptography, utilizes the same secret key to encrypt and decrypt messages.

• A key distribution center shares a (different) secret key with every user in the network; the key distribution center generates a session key to be used for a transaction.

• One of the most commonly used symmetric encryption algorithms is the Data Encryption Standard (DES).

• A block cipher is an encryption method that creates groups of bits from an original message, then applies an encryption algorithm to the block as a whole, rather than as individual bits.

• Triple DES, or 3DES, is a variant of DES that is essentially three DES systems in a row, each having its own secret key.

• The new standard for encryption is called the Advanced Encryption Standard (AES). Rijndael—a candidate for AES—is an encryption method that can be used with key sizes and block sizes of 128, 192 or 256 bits.

• Public-key cryptography is asymmetric—it uses two inversely related keys: A public key and a private key. The most commonly used public-key algorithm is RSA, an encryption system developed in 1977 by MIT professors Ron Rivest, Adi Shamir and Leonard Adleman.

• Pretty Good Privacy (PGP) is a public-key encryption system used for encrypting e-mail messages and files. PGP was designed in 1991 by Phillip Zimmermann.

• Public-key algorithms should not be thought of as a replacement for secret-key algorithms. Instead, public-key algorithms can be used to allow two parties to agree upon a key to be used for secret-key encryption over an unsecure medium.

• The process by which two parties can exchange keys over an unsecure medium is called a key agreement protocol. A protocol sets the rules for communication: Exactly what encryption algorithm(s) is (are) going to be used? The most common key agreement protocol is a digital envelope.

• Maintaining the secrecy of private keys is crucial to keeping cryptographic systems secure. Most compromises in security result from poor key management (e.g., the mishandling of private keys, resulting in key theft) rather than attacks that attempt to guess the keys.

• A main component of key management is key generation—the process by which keys are created. A malicious third party could try to decrypt a message by using every possible decryption key, a process known as brute-force cracking.

• The Java Cryptography Extension (JCE) provides Java applications with secret-key encryption, such as 3DES, and public-key algorithms, such as Diffie-Hellman and RSA.

• The JCE architecture is provider-based—developers can add new algorithms to their programs by adding a new algorithm provider.

• Class Cipher is the fundamental building block for applications that use JCE. A Cipher performs encryption and decryption using a specified algorithm (e.g., DES, 3DES, Blowfish, etc.).

• The JCE includes the SunJCE provider, which has support for several common algorithms.

• A digital signature—the electronic equivalent of written signature—authenticates the sender’s identity and, like a written signature, is difficult to forge. To create a digital signature, a sender first takes the original plaintext message and runs it through a hash function, which is a mathematical calculation that gives the message a hash value.

• The Secure Hash Algorithm (SHA-1) is the standard for hash functions. Running a message through the SHA-1 algorithm produces a 160-bit hash value.

• MD5 is another popular hash function, which was developed by Ronald Rivest to verify data integrity through a 128-bit hash value of the input file.
The hash value is also known as a message digest. The chance that two different messages will have the same message digest is statistically insignificant.

A digital certificate is a digital document that identifies a user and is issued by a certificate authority (CA). Digital certificates contain an expiration date to force users to switch key pairs. Canceled and revoked certificates are placed on a certificate revocation list (CRL). An alternative to CRLs is the Online Certificate Status Protocol (OCSP), which validates certificates in real-time. OCSP technology is currently under development.

Java provides the keytool utility for managing and generating keys, certificates and digital signatures. A keystore is a repository for storing public and private keys. An alias identifies a particular public and private key pair.

A trusted keystore is a keystore that contains certificates that the user knows to be correct, such as those from certificate authorities—to validate information with the signature.

The basis of Java security is the Java Sandbox—the protected environment in which Java applications and applets run. The Java Sandbox security model is comprised of three individual security checks: the bytecode verifier, the class loader and security manager.

Permissions may be granted on the basis of security policy files. Permissions are comprised of varying levels of access to specific resources. Any permission that is not declared explicitly in the policy file is not granted. The security policy files are external text files with certain syntax and class names.

A system-wide security policy file is responsible for granting code the permission to access files and ports on the entire system. The virtual machine loads this policy file as part of the virtual machine’s initialization. Particular applications can specify custom security policy files on the command line.

Java applets run under strict security restrictions due to the unreliability of code downloading over public networks. Developers who wish to distribute applets with special permissions must sign those applets with digital signatures.

Signing an applet with a digital signature requires that the signing party stores the applet and its supporting classes in a JAR file.

Ensuring that users actually are who they claim to be is a large part of computer security, known as authentication. Java uses the Java Authentication and Authorization Service (JAAS) for authenticating and authorizing users.

JAAS is based on a plug-in framework, which allows Kerberos and single sign-on to be implemented for authentication and authorization.

Kerberos is a freely available open-source protocol developed at MIT. It employs secret-key cryptography to authenticate users in a network and to maintain the integrity and privacy of network communications.

Java addresses the problems often associated with authenticating users and controlling access with the Java Authentication and Authorization Service (JAAS) API.

The Pluggable Authentication Module (PAM) architecture is the standard method for authentication on which JAAS is based.

After a user has been authenticated, JAAS can grant or restrict access to certain resources of an application. JAAS can control access by group, user or role-based security policies.

User-based access control governs access to resources on an individual user basis. Group-based authorization identifies a user as a part of a group and grants access to certain resources based on the identifying group. Role-based access control (RBAC) is used in addition to group-access control, allowing for more control over resources.
Chapter 7 Security 439

- The Secure Sockets Layer (SSL) protocol, developed by Netscape Communications, is a nonproprietary protocol commonly used to secure communication between two computers on the Internet and the Web. SSL operates between the Internet’s TCP/IP communications protocol and the application software.

- SSL implements public-key technology using the RSA algorithm and digital certificates to authenticate the server in a transaction and to protect private information.

- SSL is used primarily to secure point-to-point connections—transmissions of data from one computer to another. The Transport Layer Security (TLS) protocol, designed by the Internet Engineering Task Force, is similar to SSL.

- Sun’s Java Secure Socket Extension (JSSE) provides encryption, message integrity checks and authentication of the server and client.

- Before the Java Virtual Machine executes a Java program, the bytecodes for that program must pass through the bytecode verifier. The bytecode verifier ensures that the bytecodes are valid Java bytecodes that do not perform certain illegal operations.

- The Java Virtual Machine checks any remaining cast operations to ensure their validity and performs array-bounds checking to prevent programs from reading arbitrary memory locations.

- The Java Virtual Machine uses class loaders to read class definitions from .class files and produce class representations in memory.

- The class loader uses separate namespaces to prevent malicious code from interacting with safe code. Every class that the JVM loads is assigned a corresponding set of unique names, known as a namespace. Namespaces in Java act as barriers between classes.

- The bootstrap class loader loads all of the trusted core Java classes.

- The Java security manager is the module that performs security checks while code is running. If, during run-time, code attempts to execute a dangerous operation, a security exception is generated.

- Final classes cannot be subclassed, and can help make an application more secure by preventing attackers from creating malicious subclasses that can cause damage to an application or gain access to protected information.

TERMINOLOGY

Advanced Encryption Standard (AES) collision
asymmetric algorithms cryptanalysis
authentication cryptography
cipher cryptosystem
cipher block cipher DES
block-cipher Data Encryption Standard (DES)
bootstrap class loader Decorator design pattern
brute-force decryption
codebase DES cracker machine
certificate authority hierarchy Diffie-Hellman Key Agreement Protocol
certificate repository digital certificate
certificate revocation list (CRL) digital envelope
certification authority (CA) digital IDs
cipher digital signature
cipher class Diffie-Hellman Key Agreement Protocol
ciphertext encryption
class loader hacker
codebase hash function
codebase hash value
SELF-REVIEW EXERCISES

7.1 State whether the following are true or false. If the answer is false, explain why.
   a) In a public-key algorithm, one key is used for both encryption and decryption.
   b) Digital certificates are intended to be used indefinitely.
   c) Secure Sockets Layer protects data stored on a merchant’s server.
   d) Transport Layer Security is similar to the Secure Sockets Layer protocol.
   e) Digital signatures can be used to provide undeniable proof of the author of a document.
   f) In a network of 10 users communicating using public-key cryptography, only 10 keys are needed in total.
   g) The security of modern cryptosystems lies in the secrecy of the algorithm.
   h) Users should avoid changing keys as much as possible, unless they have reason to believe that the security of the key has been compromised.
   i) Increasing the security of a network often decreases its functionality and efficiency.
   j) Kerberos is an authentication protocol that is used over TCP/IP networks.
   k) SSL can be used to connect a network of computers over the Internet.
7.2 Fill in the blanks in each of the following statements:

a) Cryptographic algorithms in which the message’s sender and receiver both hold an identical key are called _________.
b) A _________ is used to authenticate the sender of a document.
c) In a _________, a document is encrypted using a symmetric secret key and sent with that symmetric secret key, encrypted using a public-key algorithm.
d) A certificate that needs to be revoked before its expiration date is placed on a _________.
e) A digital fingerprint of a document can be created using a _________.
f) The four main issues addressed by cryptography are ________, ________, ________, and ________.
g) Trying to decrypt ciphertext without knowing the decryption key is known as _________.
h) A hacker that tries every possible solution to crack a code is using a method known as _________.

ANSWERS TO SELF-REVIEW EXERCISES

7.1 a) False. The encryption key is different from the decryption key. One is made public, and the other is kept private. b) False. Digital certificates are created with an expiration date to encourage users to periodically change their public/private-key pair. c) False. Secure Sockets Layer is an Internet security protocol, which secures the transfer of information in electronic communication. It does not protect data stored on a merchant’s server. d) True. e) False. A user who digitally signed a document could later intentionally give up his or her private key and then claim that the document was written by an imposter. Thus, timestamping a document is necessary, so that users cannot repudiate documents written before the public/private-key pair is reported as invalidated. f) False. Each user needs a public key and a private key. Thus, in a network of 10 users, 20 keys are needed in total. g) False. The security of modern cryptosystems lies in the secrecy of the encryption and decryption keys. h) False. Changing keys often is a good way to maintain the security of a communication system. i) True. j) True. k) False, IPSec can connect a whole network of computers, while SSL can only connect two secure systems.

7.2 a) symmetric key algorithms. b) digital signature. c) digital envelope. d) certificate revocation list. e) hash function. f) privacy, authentication, integrity, nonrepudiation. g) cryptanalysis. h) brute-force hacking.

EXERCISES

7.3 Define each of the following security terms, and give an example of how it is used:

a) secret-key cryptography
b) public-key cryptography
c) digital signature
d) digital certificate
e) hash function
f) SSL
g) Kerberos

7.4 Write the full name and describe each of the following acronyms:

a) PKI
b) CRL
c) AES
d) SSL
List the four problems addressed by cryptography, and give a real-world example of each.

Compare symmetric-key algorithms with public-key algorithms. What are the benefits and drawbacks of each type of algorithm? How are these differences manifested in the real-world uses of the two types of algorithms?

Create a simple application that reads a file; the application should accept the file path from the command line. Install a SecurityManager to control access and create two policy files—one that grants read permission to a particular directory (e.g., a C:\ readOnly directory), and another that grants read permission to that directory for a specific codebase (e.g., the C:/myclasses codebase).

[Note: Code always has read permission to the directory in which it is running and any subdirectories.]

Using your solution to Exercise 7.7, store your application in a JAR file. Sign the JAR file with a digital signature (use the keytool to create a public/private key pair); grant read permission to the directory for code signed by this signature in a policy file. [Note: To grant permission for a particular digital signature, use the signedBy field in a grant statement. The signedBy field requires that you specify the keystore in which this public/private key pair is stored using a keytool "keystore_URL"; statement outside of the grant statements in the policy file.]

WORKS CITED
8. “MD5 Homepage (Unofficial),” <userpages.umbc.edu/~mabzug1/cs/md5/md5.html>.

BIBLIOGRAPHY
Objectives

- To understand the relational-database model.
- To understand basic database queries using Structured Query Language (SQL).
- To use the classes and interfaces of package `java.sql` to manipulate databases.
- To use transaction processing to prevent database updates from modifying the database if an error occurs during a transaction.
- To introduce the JDBC 2.0 optional package `javax.sql`'s capabilities for obtaining database connections, creating connection pools and treating result sets as JavaBeans.

*It is a capital mistake to theorize before one has data.*
Arthur Conan Doyle

*Now go, write it before them in a table, and note it in a book, that it may be for the time to come for ever and ever.*
The Holy Bible: The Old Testament

*Let's look at the record.*
Alfred Emanuel Smith

*Get your facts first, and then you can distort them as much as you please.*
Mark Twain

*I like two kinds of men: domestic and foreign.*
Mae West
8.1 Introduction

A database is an integrated collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation of the data. A database management system (DBMS) provides mechanisms for storing and organizing data in a manner consistent with the database’s format. Database management systems allow for the access and storage of data without worrying about the internal representation of databases.
Today’s most popular database systems are relational databases. A language called Structured Query Language (SQL—pronounced as its individual letters, or as “sequel”) is used almost universally with relational-database systems to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data. [Note: In this chapter, we assume that SQL is pronounced as its individual letters. For this reason, we often precede SQL with the article “an,” as in “an SQL database” or “an SQL statement.”]

Some popular enterprise-level relational-database systems are Microsoft SQL Server, Oracle, Sybase, DB2, Informix and MySQL. In this chapter, we present examples using Cloudscape 3.6.4—a pure-Java database management system from Informix Software. Cloudscape 3.6.4 is on the CD that accompanies this book and can be downloaded from www.cloudscape.com. Later chapters reuse the material presented in this chapter. Chapter 11 introduces the Java 2 Enterprise Edition reference implementation, which includes an earlier version of Cloudscape. That chapter discusses how to integrate the latest version of Cloudscape with the Java 2 Enterprise Edition. [Note: We discuss basic Cloudscape features required to execute the examples in this chapter. Please refer to the detailed Cloudscape documentation for complete information on using Cloudscape.]

Java programs communicate with databases and manipulate their data using the Java Database Connectivity (JDBC) API. A JDBC driver implements the interface to a particular database. This separation of the API from particular drivers enables developers to change the underlying database without modifying Java code that accesses the database. Most popular database management systems now include JDBC drivers. There are also many third-party JDBC drivers available. In this chapter, we introduce JDBC and use it to manipulate a Cloudscape database. The techniques demonstrated here also can be used to manipulate other databases that have JDBC drivers. Check your database management system’s documentation to determine whether your DBMS comes with a JDBC driver. Even if your DBMS does not come with a JDBC driver, many third-party vendors provide JDBC drivers for a wide variety of databases. For more information on JDBC, visit

\[\text{java.sun.com/products/jdbc/}\]

This site contains information concerning JDBC, including the JDBC specifications, FAQs on JDBC, a learning resource center, software downloads and other important information. For a list of available JDBC drivers, visit

\[\text{industry.java.sun.com/products/jdbc/drivers/}\]

This site provides a search engine to help you locate drivers appropriate to your DBMS.

### 8.2 Relational-Database Model

The relational-database model is a logical representation of data that allows the relationships between the data to be examined without consideration of the physical structure of the data. A relational database is composed of tables. Figure 8.1 illustrates a sample table that might be used in a personnel system. The table name is **Employee**, and its primary purpose is to illustrate the attributes of an employee and how they are related to a specific employee. Each row of the table is called a **record**. This table consists of six records. The **Number field** (or **column**) of each record in this table is the **primary key** for referencing data in the table. A primary key is a field (or fields) in a table that contain(s) unique data that cannot be duplicated in other records. This guarantees that each record can be identified by a unique value. Good
examples of primary key fields are a Social Security number, an employee ID number and a part number in an inventory system. The records of Fig. 8.1 are ordered by primary key. In this case, the records are listed in increasing order; we could also use decreasing order.

Each column of the table represents a different field (or column, or attribute). Records are normally unique (by primary key) within a table, but particular field values may be duplicated between records. For example, three different records in the Employee table’s Department field contain number 413.

Different users of a database often are interested in different data and different relationships among those data. Some users require only subsets of the table columns. To obtain table subsets, we use SQL statements to specify the data to select from a table. SQL provides a complete set of commands (including SELECT) that enable programmers to define complex queries that select data from a table. The results of a query are commonly called result sets (or record sets). For example, we might select data from the table in Fig. 8.1 to create a new result set that shows where departments are located. This result set is shown in Fig. 8.2. SQL queries are discussed in Section 8.4.

8.3 Relational Database Overview: The books Database

This section gives an overview of SQL in the context of a sample books database we created for this chapter. Before we discuss SQL, we overview the tables of the books database. We use this to introduce various database concepts, including the use of SQL to obtain useful information from the database and to manipulate the database. We provide a script to create the database. You can find the script in the examples directory for this chapter on the CD that accompanies this book. Section 8.5 explains how to use this script.
The database consists of four tables: **authors**, **publishers**, **authorISBN** and **titles**. The **authors** table (described in Fig. 8.3) consists of three fields (or columns) that maintain each author’s unique ID number, first name and last name. Figure 8.4 contains the data from the **authors** table of the **books** database.

The **publishers** table (described in Fig. 8.5) consists of two fields representing each publisher’s unique ID and name. Figure 8.6 contains the data from the **publishers** table of the **books** database.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>authorID</strong></td>
<td>Author’s ID number in the database. In the <strong>books</strong> database, this integer field is defined as an <em>autoincremented field</em>. For each new record inserted in this table, the database automatically increments the <strong>authorID</strong> value to ensure that each record has a unique <strong>authorID</strong>. This field represents the table’s primary key.</td>
</tr>
<tr>
<td><strong>firstName</strong></td>
<td>Author’s first name (a string).</td>
</tr>
<tr>
<td><strong>lastName</strong></td>
<td>Author’s last name (a string).</td>
</tr>
</tbody>
</table>

**Fig. 8.3**  **authors** table from **books**.

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harvey</td>
<td>Deitel</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>Deitel</td>
</tr>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
<tr>
<td>4</td>
<td>Sean</td>
<td>Santry</td>
</tr>
</tbody>
</table>

**Fig. 8.4** Data from the **authors** table of **books**.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>publisherID</strong></td>
<td>The publisher’s ID number in the database. This autoincremented integer is the table’s primary-key field.</td>
</tr>
<tr>
<td><strong>publisherName</strong></td>
<td>The name of the publisher (a string).</td>
</tr>
</tbody>
</table>

**Fig. 8.5**  **publishers** table from **books**.

<table>
<thead>
<tr>
<th>publisherID</th>
<th>publisherName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Prentice Hall</td>
</tr>
<tr>
<td>2</td>
<td>Prentice Hall PTG</td>
</tr>
</tbody>
</table>

**Fig. 8.6** Data from the **publishers** table of **books**.
The **authorISBN** table (described in Fig. 8.7) consists of two fields that maintain each ISBN number and its corresponding author’s ID number. This table helps associate the names of the authors with the titles of their books. Figure 8.8 contains the data from the **authorISBN** table of the **books** database. ISBN is an abbreviation for “International Standard Book Number”—a numbering scheme that publishers worldwide use to give every book a unique identification number. [Note: To save space, we have split the contents of this table into two columns, each containing the **authorID** and **isbn** fields.]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>authorID</strong></td>
<td>The author’s ID number, which allows the database to associate each book with a specific author. The integer ID number in this field must also appear in the <strong>authors</strong> table.</td>
</tr>
<tr>
<td><strong>isbn</strong></td>
<td>The ISBN number for a book (a string).</td>
</tr>
</tbody>
</table>

**Fig. 8.7** **authorISBN** table from **books**.

<table>
<thead>
<tr>
<th>authorID</th>
<th>isbn</th>
<th>authorID</th>
<th>isbn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0130895725</td>
<td>2</td>
<td>0139163050</td>
</tr>
<tr>
<td>1</td>
<td>0132261197</td>
<td>2</td>
<td>013028419x</td>
</tr>
<tr>
<td>1</td>
<td>0130895717</td>
<td>2</td>
<td>0130161438</td>
</tr>
<tr>
<td>1</td>
<td>0135289106</td>
<td>2</td>
<td>0130856118</td>
</tr>
<tr>
<td>1</td>
<td>0139163050</td>
<td>2</td>
<td>0130125075</td>
</tr>
<tr>
<td>1</td>
<td>013028419x</td>
<td>2</td>
<td>0138993947</td>
</tr>
<tr>
<td>1</td>
<td>0130161438</td>
<td>2</td>
<td>0130852473</td>
</tr>
<tr>
<td>1</td>
<td>0130856118</td>
<td>2</td>
<td>0130829277</td>
</tr>
<tr>
<td>1</td>
<td>0130125075</td>
<td>2</td>
<td>0134569555</td>
</tr>
<tr>
<td>1</td>
<td>0138993947</td>
<td>2</td>
<td>0130829293</td>
</tr>
<tr>
<td>1</td>
<td>0130852473</td>
<td>2</td>
<td>0130284173</td>
</tr>
<tr>
<td>1</td>
<td>0130829277</td>
<td>2</td>
<td>0130284181</td>
</tr>
<tr>
<td>1</td>
<td>0134569555</td>
<td>2</td>
<td>0130895601</td>
</tr>
<tr>
<td>1</td>
<td>0130829293</td>
<td>3</td>
<td>013028419x</td>
</tr>
<tr>
<td>1</td>
<td>0130284173</td>
<td>3</td>
<td>0130161438</td>
</tr>
<tr>
<td>1</td>
<td>0130284181</td>
<td>3</td>
<td>0130856118</td>
</tr>
<tr>
<td>1</td>
<td>0130895601</td>
<td>3</td>
<td>0134569555</td>
</tr>
<tr>
<td>2</td>
<td>0130895725</td>
<td>3</td>
<td>0130829293</td>
</tr>
<tr>
<td>2</td>
<td>0132261197</td>
<td>3</td>
<td>0130284173</td>
</tr>
<tr>
<td>2</td>
<td>0130895717</td>
<td>3</td>
<td>0130284181</td>
</tr>
<tr>
<td>2</td>
<td>0135289106</td>
<td>4</td>
<td>0130895601</td>
</tr>
</tbody>
</table>
The **titles** table (described in Fig. 8.9) consists of six fields that maintain general information about each book in the database, including the ISBN number, title, edition number, copyright year, publisher’s ID number, name of a file containing an image of the book cover, and finally, the price. Figure 8.10 contains the data from the **titles** table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn</td>
<td>ISBN number of the book (a string).</td>
</tr>
<tr>
<td>title</td>
<td>Title of the book (a string).</td>
</tr>
<tr>
<td>copyright</td>
<td>Copyright year of the book (a string).</td>
</tr>
<tr>
<td>publisherID</td>
<td>Publisher's ID number (an integer). This value must correspond to an ID number in the <strong>publishers</strong> table.</td>
</tr>
<tr>
<td>imageFile</td>
<td>Name of the file containing the book’s cover image (a string).</td>
</tr>
</tbody>
</table>
| price      | Suggested retail price of the book (a real number). [Note: The prices shown in this book are for example purposes only.]

---

**Fig. 8.9**  **titles** table from **books**.

<table>
<thead>
<tr>
<th>isbn</th>
<th>title</th>
<th>editionNumber</th>
<th>copyright</th>
<th>publisherID</th>
<th>imageFile</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>0130895725</td>
<td>C How to Program</td>
<td>3</td>
<td>2001</td>
<td>1</td>
<td>chtp3.jpg</td>
<td>69.95</td>
</tr>
<tr>
<td>0132261197</td>
<td>C How to Program</td>
<td>2</td>
<td>1994</td>
<td>1</td>
<td>chtp2.jpg</td>
<td>49.95</td>
</tr>
<tr>
<td>0130895717</td>
<td>C++ How to Program</td>
<td>3</td>
<td>2001</td>
<td>1</td>
<td>cpphtp3.jpg</td>
<td>69.95</td>
</tr>
<tr>
<td>0135289106</td>
<td>C++ How to Program</td>
<td>2</td>
<td>1998</td>
<td>1</td>
<td>cpphtp2.jpg</td>
<td>49.95</td>
</tr>
<tr>
<td>0139163050</td>
<td>The Complete C++ Training Course</td>
<td>3</td>
<td>2001</td>
<td>2</td>
<td>cppctc3.jpg</td>
<td>109.95</td>
</tr>
<tr>
<td>013028419x</td>
<td>e-Business and e-Commerce How to Program</td>
<td>1</td>
<td>2001</td>
<td>1</td>
<td>ebechtpl.jpg</td>
<td>69.95</td>
</tr>
<tr>
<td>0130161438</td>
<td>Internet and World Wide Web How to Program</td>
<td>1</td>
<td>2000</td>
<td>1</td>
<td>iw3htpl.jpg</td>
<td>69.95</td>
</tr>
<tr>
<td>0130856118</td>
<td>The Complete Internet and World Wide Web Programming Training Course</td>
<td>1</td>
<td>2000</td>
<td>2</td>
<td>iw3ctc1.jpg</td>
<td>109.95</td>
</tr>
<tr>
<td>0130125075</td>
<td>Java How to Program (Java 2)</td>
<td>3</td>
<td>2000</td>
<td>1</td>
<td>jhtp3.jpg</td>
<td>69.95</td>
</tr>
<tr>
<td>0138993947</td>
<td>Java How to Program (Java 1.1)</td>
<td>2</td>
<td>1998</td>
<td>1</td>
<td>jhtp2.jpg</td>
<td>49.95</td>
</tr>
</tbody>
</table>

---

**Fig. 8.10**  Data from the **titles** table of **books** (part 1 of 2).
Figure 8.11 illustrates the relationships among the tables in the books database. The first line in each table is the table’s name. The field name in green contains that table’s primary key. A table’s primary key uniquely identifies each record in the table. Every record must have a value in the primary-key field, and the value must be unique. This is known as the Rule of Entity Integrity.

**Common Programming Error 8.1**

Not providing a value for a primary-key field in every record breaks the Rule of Entity Integrity and causes the DBMS to report an error.

**Common Programming Error 8.2**

Providing duplicate values for the primary-key field in multiple records causes the DBMS to report an error.
The lines connecting the tables in Fig. 8.11 represent the relationships between the tables. Consider the line between the publishers and titles tables. On the publishers end of the line, there is a 1, and on the titles end, there is an infinity (∞) symbol, indicating a one-to-many relationship in which every publisher in the publishers table can have an arbitrarily large number of books in the titles table. Note that the relationship line links the publisherID field in the table publishers to the publisherID field in table titles. The publisherID field in the titles table is a foreign key—a field for which every entry has a unique value in another table and where the field in the other table is the primary key for that table (e.g., publisherID in the publishers table). Foreign keys are specified when creating a table. The foreign key helps maintain the Rule of Referential Integrity: Every foreign key-field value must appear in another table’s primary-key field. Foreign keys enable information from multiple tables to be joined together for analysis purposes. There is a one-to-many relationship between a primary key and its corresponding foreign key. This means that a foreign key-field value can appear many times in its own table, but can only appear once as the primary key of another table. The line between the tables represents the link between the foreign key in one table and the primary key in another table.

Common Programming Error 8.3

Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity and causes the DBMS to report an error.

The line between the authorISBN and authors tables indicates that for each author in the authors table, there can be an arbitrary number of ISBNs for books written by that author in the authorISBN table. The authorID field in the authorISBN table is a foreign key of the authorID field (the primary key) of the authors table. Note again that the line between the tables links the foreign key of table authorISBN to the corresponding primary key in table authors. The authorISBN table links information in the titles and authors tables.

Finally, the line between the titles and authorISBN tables illustrates a one-to-many relationship; a title can be written by any number of authors. In fact, the sole purpose of the authorISBN table is to represent a many-to-many relationship between the authors and titles tables; an author can write any number of books and a book can have any number of authors.

8.4 Structured Query Language (SQL)

In this section, we provide an overview of SQL in the context of our books sample database. You will be able to use the SQL queries discussed here in the examples later in the chapter.

The SQL keywords listed in Fig. 8.12 are discussed in the context of complete SQL queries in the next several subsections; other SQL keywords are beyond the scope of this text. [Note: For more information on SQL, please refer to the World Wide Web resources in Section 8.13 and the bibliography at the end of this chapter.]
8.4.1 Basic SELECT Query

Let us consider several SQL queries that extract information from database books. A typical SQL query “selects” information from one or more tables in a database. Such selections are performed by SELECT queries. The simplest format of a SELECT query is

```
SELECT * FROM tableName
```

In this query, the asterisk (*) indicates that all rows and columns from the tableName table of the database should be selected. For example, to select the entire contents of the authors table (i.e., all the data in Fig. 8.4), use the query

```
SELECT * FROM authors
```

To select specific fields from a table, replace the asterisk (*) with a comma-separated list of the field names to select. For example, to select only the fields authorID and lastName for all rows in the authors table, use the query

```
SELECT authorID, lastName FROM authors
```

This query returns the data listed in Fig. 8.13. [Note: If a field name contains spaces, it must be enclosed in square brackets ([]) in the query. For example, if the field name is firstName, the field name would appear in the query as [firstName].]
Software Engineering Observation 8.1

For most SQL statements, the asterisk (*) should not be used to specify field names to select from a table (or several tables). In general, programmers process result sets by knowing in advance the order of the fields in the result set. For example, selecting \texttt{authorID} and \texttt{lastName} from table \texttt{authors} guarantees that the fields will appear in the result set with \texttt{authorID} as the first field and \texttt{lastName} as the second field. As you will see, programs typically process result set fields by specifying the column number in the result set (column numbers start at 1 for the first field in the result set).

Software Engineering Observation 8.2

Specifying the field names to select from a table (or several tables) guarantees that the fields are always returned in the specified order and also avoid returning unused fields, even if the actual order of the fields in the database table(s) changes.

Common Programming Error 8.4

If a programmer assumes that the fields in a result set are always returned in the same order from an SQL statement that uses the asterisk (*) to select fields, the program may process the result set incorrectly. If the field order in the database table(s) changes, the order of the fields in the result set would change accordingly.

Performance Tip 8.1

If the order of fields in a result set is unknown to a program, the program must process the fields by name. This can require a linear search of the field names in the result set. Specifying the field names to select from a table (or several tables) enables the application receiving the result set to know the order of the fields in advance. In this case, the program can process the data more efficiently, because fields can be accessed directly by column number.

8.4.2 WHERE Clause

In most cases, it is necessary to locate records in a database that satisfy certain selection criteria. Only records that match the selection criteria are selected. SQL uses the optional \texttt{WHERE} clause in a \texttt{SELECT} query to specify the selection criteria for the query. The simplest format of a \texttt{SELECT} query with selection criteria is

\begin{verbatim}
SELECT fieldName1, fieldName2, ... FROM tableName WHERE criteria
\end{verbatim}

For example, to select the \texttt{title}, \texttt{editionNumber} and \texttt{copyright} fields from table \texttt{titles} for which the \texttt{copyright} date is greater than 1999, use the query

\begin{verbatim}
SELECT title, editionNumber, copyright
FROM titles
WHERE copyright > 1999
\end{verbatim}

Figure 8.14 shows the results of the preceding query. [Note: When we construct a query for use in Java, we simply create a \texttt{String} containing the entire query. When we display queries in the text, we often use multiple lines and indentation for readability.]

Performance Tip 8.2

Using selection criteria improves performance by selecting a portion of the database that is normally smaller than the entire database. Working with a smaller portion of the data is more efficient than working with the entire set of data stored in the database.
The **WHERE** clause condition can contain operators `<`, `>`, `<=`, `>=`, `=` and **LIKE**. Operator **LIKE** is used for *pattern matching* with wildcard characters *percent* (`%`) and *underscore* (`_`). Pattern matching allows SQL to search for similar strings that match a given pattern.

A pattern that contains a percent character (`%`) searches for strings that have zero or more characters at the percent character’s position in the pattern. For example, the following query locates the records of all the authors whose last name starts with the letter D:

```sql
SELECT authorID, firstName, lastName
FROM authors
WHERE lastName LIKE 'D%'
```

The preceding query selects the two records shown in Fig. 8.15, because two of the four authors in our database have a last name starting with the letter D (followed by zero or more characters). The `%` in the **WHERE** clause’s **LIKE** pattern indicates that any number of characters can appear after the letter D in the `lastName` field. Notice that the pattern string is surrounded by single-quote characters.
Portability Tip 8.1
See the documentation for your database system to determine whether SQL is case sensitive on your system and to determine the syntax for SQL keywords (i.e., should they be all uppercase letters, all lowercase letters or some combination of the two?).

Portability Tip 8.2
Not all database systems support the LIKE operator, so be sure to read your database system’s documentation carefully.

Portability Tip 8.3
Some databases use the * character in place of the % character in a LIKE expression.

Portability Tip 8.4
In some databases (including Cloudscape), string data is case sensitive.

Good Programming Practice 8.1
By convention, SQL keywords should use all uppercase letters on systems that are not case sensitive, to emphasize the SQL keywords in an SQL statement.

An underscore (_ ) in the pattern string indicates a single character at that position in the pattern. For example, the following query locates the records of all the authors whose last name starts with any character (specified by _), followed by the letter i, followed by any number of additional characters (specified by %):

```
SELECT authorID, firstName, lastName
FROM authors
WHERE lastName LIKE '_i%'
```

The preceding query produces the record shown in Fig. 8.16, because only one author in our database has a last name that contains the letter i as its second letter.

Portability Tip 8.5
Some databases use the _ character in place of the ? character in a LIKE expression.

8.4.3 ORDER BY Clause
The results of a query can be arranged in ascending or descending order by using the optional ORDER BY clause. The simplest form of an ORDER BY clause is

```
SELECT fieldName1, fieldName2, ... FROM tableName ORDER BY field ASC
SELECT fieldName1, fieldName2, ... FROM tableName ORDER BY field DESC
```

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
</tbody>
</table>

Fig. 8.16 The only author from the authors table whose last name contains i as the second letter.
where **ASC** specifies ascending order (lowest to highest), **DESC** specifies descending order (highest to lowest) and **field** specifies the field on which the sort is based.

For example, to obtain the list of authors in ascending order by last name (Fig. 8.17), use the query

```sql
SELECT authorID, firstName, lastName
FROM authors
ORDER BY lastName ASC
```

Note that the default sorting order is ascending, so **ASC** is optional. To obtain the same list of authors in descending order by last name (Fig. 8.18), use the query

```sql
SELECT authorID, firstName, lastName
FROM authors
ORDER BY lastName DESC
```

Multiple fields can be used for ordering purposes with an **ORDER BY** clause of the form

```
ORDER BY field1 sortingOrder, field2 sortingOrder, ...
```

where **sortingOrder** is either **ASC** or **DESC**. Note that the **sortingOrder** does not have to be identical for each field. The query

```sql
SELECT authorID, firstName, lastName
FROM authors
ORDER BY lastName, firstName
```

sorts in ascending order all the authors by last name, then by first name. If any authors have the same last name, their records are returned in sorted order by their first name (Fig. 8.19).

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Paul</td>
<td>Deitel</td>
</tr>
<tr>
<td>1</td>
<td>Harvey</td>
<td>Deitel</td>
</tr>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
<tr>
<td>4</td>
<td>Sean</td>
<td>Santry</td>
</tr>
</tbody>
</table>

**Fig. 8.17** Authors from table **authors** in ascending order by **lastName**.

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Sean</td>
<td>Santry</td>
</tr>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>Deitel</td>
</tr>
<tr>
<td>1</td>
<td>Harvey</td>
<td>Deitel</td>
</tr>
</tbody>
</table>

**Fig. 8.18** Authors from table **authors** in descending order by **lastName**.
The **WHERE** and **ORDER BY** clauses can be combined in one query. For example, the query

```
SELECT isbn, title, editionNumber, copyright, price
FROM titles
WHERE title
LIKE '%How to Program'
ORDER BY title ASC
```

returns the **isbn**, **title**, **editionNumber**, **copyright** and **price** of each book in the **titles** table that has a **title** ending with **"How to Program"** and orders them in ascending order by **title**. The results of the query are shown in Fig. 8.20. Note that the title “e-Business and e-Commerce How to Program” appears at the end of the list, because Cloudscape uses the Unicode numeric values of the characters for comparison purposes. Remember that lowercase letters have larger numeric values than uppercase letters.
8.4.4 Merging Data from Multiple Tables: Joining

Often, it is necessary to merge data from multiple tables into a single view for analysis purposes. This is referred to as joining the tables and is accomplished by using a comma-separated list of tables in the `FROM` clause of a `SELECT` query. A join merges records from two or more tables by testing for matching values in a field that is common to both tables. The simplest format of a join is

```
SELECT fieldName1, fieldName2, ...
FROM table1, table2
WHERE table1.fieldName = table2.fieldName
```

The query’s `WHERE` clause specifies the fields from each table that should be compared to determine which records will be selected. These fields normally represent the primary key in one table and the corresponding foreign key in the other table. For example, the following query produces a list of authors and the ISBN numbers for the books that each author wrote:

```
SELECT firstName, lastName, isbn
FROM authors, authorISBN
WHERE authors.authorID = authorISBN.authorID
ORDER BY lastName, firstName
```

The query merges the `firstName` and `lastName` fields from table `authors` and the `isbn` field from table `authorISBN` and sorts the results in ascending order by `lastName` and `firstName`. Notice the use of the syntax `tableName.fieldName` in the `WHERE` clause of the query. This syntax (called a fully qualified name) specifies the fields from each table that should be compared to join the tables. The “`tableName`.” syntax is required if the fields have the same name in both tables.

**Software Engineering Observation 8.3**

If an SQL statement uses fields with the same name from multiple tables, the field name must be fully qualified with its table name and a dot operator (.), as in `authors.authorID`.

**Common Programming Error 8.5**

In a query, not providing fully qualified names for fields with the same name from two or more tables is an error.

As always, the `FROM` clause can be followed by an `ORDER BY` clause. Figure 8.21 shows the results of the preceding query. [Note: To save space, we split the results of the query into two columns, each containing the `firstName`, `lastName` and `isbn` fields.]

<table>
<thead>
<tr>
<th>firstName</th>
<th>lastName</th>
<th>isbn</th>
<th>firstName</th>
<th>lastName</th>
<th>isbn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harvey</td>
<td>Deitel</td>
<td>0130895601</td>
<td>Harvey</td>
<td>Deitel</td>
<td>0130284173</td>
</tr>
<tr>
<td>Harvey</td>
<td>Deitel</td>
<td>0130284181</td>
<td>Harvey</td>
<td>Deitel</td>
<td>0130829293</td>
</tr>
</tbody>
</table>

**Fig. 8.21** Authors and the ISBN numbers for the books they have written in ascending order by `lastName` and `firstName` (part 1 of 2).
8.4.5 INSERT INTO Statement

The `INSERT INTO` statement inserts a new record into a table. The simplest form of this statement is

```
INSERT INTO tableName ( fieldName1, fieldName2, ..., fieldNameN )
VALUES ( value1, value2, ..., valueN )
```

where `tableName` is the table in which to insert the record. The `tableName` is followed by a comma-separated list of field names in parentheses (this list is not required if the `INSERT INTO` operation specifies a value for every column of the table in the correct order). The list of field names is followed by the SQL keyword `VALUES` and a comma-separated list of values in parentheses. The values specified here should match the field names specified after the table name in order and type (i.e., if `fieldName1` is supposed to be the `firstName` field, then `value1` should be a string in single quotes representing the first name). Always use the list of field names when inserting new records. If the order of the fields changes in the table, entering only `VALUES` may cause an error. The `INSERT INTO` statement

```
INSERT INTO authors ( firstName, lastName )
VALUES ( 'Sue', 'Smith' )
```

Fig. 8.21 Authors and the ISBN numbers for the books they have written in ascending order by `lastName` and `firstName` (part 2 of 2).
inserts a record into the authors table. The statement indicates that values will be inserted for the firstName and lastName fields. The corresponding values to insert are 'Sue' and 'Smith'. We do not specify an authorID in this example, because authorID is an autoincremented field in the Cloudscape database. For every new record added to this table, Cloudscape assigns a unique authorID value that is the next value in the autoincremented sequence (i.e., 1, 2, 3 and so on). In this case, Sue Smith would be assigned authorID number 5. Figure 8.22 shows the authors table after the INSERT INTO operation.

Common Programming Error 8.6

It is an error to specify a value for an autoincrement field.

Common Programming Error 8.7

SQL statements use the single-quote (‘) character as a delimiter for strings. To specify a string containing a single quote (such as O’Malley) in an SQL statement, the string must have two single quotes in the position where the single-quote character appears in the string (e.g., ‘O’‘Malley’). The first of the two single-quote characters acts as an escape character for the second. Not escaping single-quote characters in a string that is part of an SQL statement is an SQL syntax error.

8.4.6 UPDATE Statement

An UPDATE statement modifies data in a table. The simplest form for an UPDATE statement is

\[
\text{UPDATE } \text{tableName} \\
\text{SET } \text{fieldName1} = \text{value1}, \text{fieldName2} = \text{value2}, \ldots, \text{fieldNameN} = \text{valueN} \\
\text{WHERE } \text{criteria}
\]

where tableName is the table in which to update a record (or records). The tableName is followed by keyword SET and a comma-separated list of field name/value pairs in the format fieldName = value. The WHERE clause provides the criteria that specify which record(s) to update. The UPDATE statement

\[
\text{UPDATE } \text{authors} \\
\text{SET } \text{lastName} = 'Jones' \\
\text{WHERE } \text{lastName} = 'Smith' \ \text{AND} \ \text{firstName} = 'Sue'
\]

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harvey</td>
<td>Deitel</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>Deitel</td>
</tr>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
<tr>
<td>4</td>
<td>Sean</td>
<td>Santry</td>
</tr>
<tr>
<td>5</td>
<td>Sue</td>
<td>Smith</td>
</tr>
</tbody>
</table>

Fig. 8.22 Table Authors after an INSERT INTO operation to add a record.
updates a record in the `authors` table. The statement indicates that the `lastName` will be assigned the value `Jones` for the record in which `lastName` is equal to `Smith` and `firstName` is equal to `Sue`. [Note: If there are multiple records with the first name “Sue” and the last name “Smith,” this statement will modify all such records to have the last name “Jones.”] If we know the `authorID` in advance of the UPDATE operation (possibly because we searched for the record previously), the WHERE clause could be simplified as follows:

```
WHERE AuthorID = 5
```

Figure 8.23 shows the `authors` table after the UPDATE operation has taken place.

### 8.4.7 DELETE FROM Statement

An SQL DELETE statement removes data from a table. The simplest form for a DELETE statement is

```
DELETE FROM tableName WHERE criteria
```

where `tableName` is the table from which to delete a record (or records). The WHERE clause specifies the criteria used to determine which record(s) to delete. The DELETE statement

```
DELETE FROM authors
WHERE lastName = 'Jones' AND firstName = 'Sue'
```

deletes the record for Sue Jones in the `authors` table. If we know the `authorID` in advance of the DELETE operation, the WHERE clause could be simplified as follows:

```
WHERE authorID = 5
```

Figure 8.24 shows the `authors` table after the DELETE operation has taken place.

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harvey</td>
<td>Deitel</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>Deitel</td>
</tr>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
<tr>
<td>4</td>
<td>Sean</td>
<td>Santry</td>
</tr>
<tr>
<td>5</td>
<td>Sue</td>
<td>Jones</td>
</tr>
</tbody>
</table>

Fig. 8.23 Table authors after an UPDATE operation to change a record.

<table>
<thead>
<tr>
<th>authorID</th>
<th>firstName</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harvey</td>
<td>Deitel</td>
</tr>
<tr>
<td>2</td>
<td>Paul</td>
<td>Deitel</td>
</tr>
<tr>
<td>3</td>
<td>Tem</td>
<td>Nieto</td>
</tr>
<tr>
<td>4</td>
<td>Sean</td>
<td>Santry</td>
</tr>
</tbody>
</table>

Fig. 8.24 Table authors after a DELETE operation to remove a record.
8.5 Creating Database books in Cloudscape

The CD that accompanies this book includes Cloudscape 3.6.4, a pure-Java database management system from Informix Software. A complete set of information on Cloudscape is available from www.cloudscape.com. Follow the provided instructions to install Cloudscape. Cloudscape executes on many platforms, including Windows, Solaris, Linux, Macintosh and others. For a complete list of platforms on which Cloudscape 3.6 has been tested, visit

cloudweb1.cloudscape.com/support/servepage.jsp?page=fyi_cert36vms.html

The Cloudscape server must be executing to create and manipulate databases in Cloudscape. To execute the server, begin by opening a command window. Change directories to the Cloudscape installation directory (Cloudscape_3.6 by default). In that directory is a frameworks directory. Cloudscape comes with two frameworks in which it can execute: embedded and RmiJdbc. The embedded framework enables Cloudscape to execute as part of a Java application. The RmiJdbc framework enables Cloudscape to execute as a stand-alone database server, which is how we use Cloudscape in this book. Each framework directory has a bin subdirectory containing batch files (Windows) and shell scripts (Linux/UNIX) to set environment variables and execute Cloudscape. Change directories to the bin directory in the RmiJdbc framework. Execute the batch file or shell script starting with the name setServerCloudscapeCP to set the environment variables required by the server. Then execute the batch file or shell script starting with the name startCS to launch the Cloudscape database server. Figure 8.25 shows the command-line output when Cloudscape is executed from a Windows 2000 command window. Note that you can shut down the server by executing the script stopCS from another command window.

For each Cloudscape database we discuss in this book, we provide an SQL script that will set up the database and its tables. These scripts can be executed with an interactive command line tool, called ij, that is part of Cloudscape. We provide a batch file (createDatabase.bat) and a shell script (createDatabase.ksh) that you can use to start ij and execute the SQL scripts. In the examples directory for this chapter on the CD that accompanies this book, you will find the createDatabase scripts and the SQL script books.sql. To create database books, first ensure that the Cloudscape server is executing. Open a new command prompt, then change to Cloudscape’s frameworks\RmiJdbc\bin directory. In that directory, execute the batch file or shell script starting with the name setClientCloudscapeCP. This sets the environment variables required by our createDatabase script. Next, change to the directory where you placed our JDBC examples on your computer and type

createDatabase books.sql

to execute the SQL script. After completing this task, you are ready to proceed to the first JDBC example. [Note: We wrote this script such that you can execute the script again at any time to restore the database’s original contents. When you run this script the first time, it will generate four error messages as it tries to delete the four tables in the books database. This occurs because the database does not exist, so there are no tables to delete. You can simply ignore these messages.]
In this section, we present two examples that introduce how to connect to a database, query the database and display the results of the query.

### 8.6.1 Connecting to and Querying a JDBC Data Source

The first example (Fig. 8.26) performs a simple query on the **books** database that retrieves the entire **authors** table and displays the data in a **JTextArea**. The program illustrates connecting to the database, querying the database and processing the results. The following discussion presents the key JDBC aspects of the program. [Note: Section 8.5 demonstrates how to execute the Cloudscape database server and how to create the **books** database. The steps in Section 8.5 must be performed before executing the program of Fig. 8.26.]

```java
// Fig. 8.26: DisplayAuthors.java
// Displaying the contents of table authors in database books.
package com.deitel.advjhtp1.jdbc;

// Java core packages
import java.awt.*;
import java.sql.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

public class DisplayAuthors extends JFrame {
    // constructor connects to database, queries database,
    // processes results and displays results in window
    public DisplayAuthors() {
        super( "Authors Table of Books Database" );
    }
}
```

Fig. 8.25 Executing Cloudscape from a command prompt in Windows 2000.

Fig. 8.26 Displaying the **authors** table from the **books** database (part 1 of 3).
// connect to database books and query database
try {

// load database driver class
Class.forName( "COM.cloudscape.core.RmiJdbcDriver" );

// connect to database
Connection connection = DriverManager.getConnection(
    "jdbc:cloudscape:rmi:books" );

// create Statement to query database
Statement statement = connection.createStatement();

// query database
ResultSet resultSet =
    statement.executeQuery( "SELECT * FROM authors" );

// process query results
StringBuilder results = new StringBuffer();
ResultSetMetaData metaData = resultSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();

for ( int i = 1; i <= numberOfColumns; i++ ) {
    results.append( metaData.getColumnName( i )
    + "\t" );
}
results.append( "\n" );

while ( resultSet.next() ) {
    for ( int i = 1; i <= numberOfColumns; i++ ) {
        results.append( resultSet.getObject( i )
        + "\t" );
    }
    results += "\n";
}

// close statement and connection
statement.close();
connection.close();

// set up GUI and display window
JTextArea textArea = new JTextArea( results.toString() );
Container container = getContentPane();
container.add( new JScrollPane( textArea ) );
container.setSize( 300, 100 );  // set window size
container.setVisible( true );   // display window
}  // end try

Fig. 8.26 Displaying the authors table from the books database (part 2 of 3).
Line 7 imports package `java.sql`, which contains classes and interfaces for the JDBC API. The `DisplayAuthors` constructor (lines 17–93) connects to the books database, queries the database, displays the results of the query and closes the database connection.

The program must load the database driver class before the program can connect to the database. Line 26 loads the class definition for the database driver. This line throws a checked exception of type `java.lang.ClassNotFoundException` if the class loader cannot locate the driver class. Notice that the statement specifies the complete package name and class name for the Cloudscape driver—`COM.cloudscape.core.RmiJdbcDriver`.

JDBC supports four categories of drivers: **JDBC-to-ODBC bridge driver** (Type 1), **Native-API, partly Java driver** (Type 2); **JDBC-Net pure Java driver** (Type 3) and **Native-Protocol pure Java driver** (Type 4). A description of each driver type is shown in Fig. 8.27.
Type 3 and 4 drivers are preferred, because they are pure Java solutions. As mentioned in Fig. 8.27, Type 1 and Type 2 drivers were provided primarily to allow Java programmers to create data-driven solutions before the database vendors created pure Java drivers. The Cloudscape driver `COM.cloudscape.core.RmiJdbcDriver` is a Type 4 driver.

### Software Engineering Observation 8.4

Most major database vendors provide their own JDBC database drivers, and many third-party vendors provide JDBC drivers as well. For more information on JDBC drivers, visit the Sun Microsystems JDBC Web site, [java.sun.com/products/jdbc](http://java.sun.com/products/jdbc).

### Software Engineering Observation 8.5

On the Microsoft Windows platform, most databases support access via Open Database Connectivity (ODBC). ODBC is a technology developed by Microsoft to allow generic access to disparate database systems on the Windows platform (and some UNIX platforms). The Java 2 Software Development Kit (J2SDK) comes with the JDBC-to-ODBC-bridge database driver to allow any Java program to access any ODBC data source. The driver is defined by class `JdbcOdbcDriver` in package `sun.jdbc.odbc`.

Lines 29–30 of Fig. 8.26 declare and initialize a `Connection` reference (package `java.sql`) called `connection`. An object that implements interface `Connection` manages the connection between the Java program and the database. `Connection` objects enable programs to create SQL statements that manipulate databases and to perform transaction processing (discussed later in this chapter). The program initializes `connection` with the result of a call to `static` method `getConnection` of class `DriverManager` (package `java.sql`), which attempts to connect to the database specified by its URL argument. The URL helps the program locate the database (possibly on a network or in the local file system of the computer). The URL `jdbc:cloudscape:rmi:books`
specifies the protocol for communication (jdbc), the subprotocol for communication (cloudscape:rmi) and the name of the database (books). The subprotocol cloudscape:rmi indicates that the program uses jdbc to connect to a Cloudscape database via remote method invocation (RMI). [Note: Knowledge of the RMI networking technology is not required for this chapter. We discuss RMI in Chapter 13.] If the DriverManager cannot connect to the database, method getConnection throws an SQLException (package java.sql).

Software Engineering Observation 8.6

Most database management systems require the user to log in before accessing the database contents. DriverManager method getConnection is overloaded with versions that enable the program to supply the user name and password to gain access.

Line 33 invokes Connection method createStatement to obtain an object that implements interface Statement (package java.sql). The program uses the Statement object to submit SQL statements to the database.

Lines 36–37 use the Statement object’s executeQuery method to execute a query that selects all the author information from table authors. This method returns an object that implements interface ResultSet and contains the results of the query. The ResultSet methods enable the program to manipulate the query results.

Lines 30–59 process the ResultSet. Line 41 obtains the metadata for the ResultSet and assigns it to a ResultSetMetaData (package java.sql) reference. The metadata describes the ResultSet’s contents. Programs can use metadata programmatically to obtain information about the ResultSet’s column names and types. Line 42 uses ResultSetMetaData method getColumnCount to retrieve the number of columns in the ResultSet. Lines 44–47 append the column names to the StringBuffer results.

Software Engineering Observation 8.7

Metadata enables programs to process ResultSet contents dynamically when detailed information about the ResultSet is not known in advance of a query.

Lines 51–59 append the data in each ResultSet row to the StringBuffer results. Before processing the ResultSet, the program positions the ResultSet cursor to the first record in the ResultSet with method next (line 51). The cursor keeps track of the current record (or row). Method next returns boolean value true if it is able to position to the next record; otherwise the method returns false.

Common Programming Error 8.8

Initially, a ResultSet cursor is positioned before the first record. Attempting to access a ResultSet’s contents before positioning the ResultSet cursor to the first record with method next causes an SQLException.

If there are records in the ResultSet, lines 53–56 extract the contents of the current row. When processing a ResultSet, it is possible to extract each column of the ResultSet as a specific Java data type. In fact, ResultSetMetaData method getColumnType returns a constant integer from class Types (package java.sql) that indicates the type of the data for a specific column. Programs can use such information in a switch structure to invoke a ResultSet method that returns the column value as the appropriate Java data type. For example, if the type of a column is Types.INT,
**ResultSet** method `getInt` returns the column value as an `int`. **ResultSet** get methods typically receive as an argument either a `column number` (as an `int`) or a `column name` (as a `String`) indicating which column’s value to obtain. Visit

```
java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html
```

for detailed mappings of SQL types to Java types and to determine the appropriate **ResultSet** method to call for each SQL type.

**Performance Tip 8.3**

If a query specifies the exact fields to select from the database, the **ResultSet** contains the fields in the specified order. In this case, using the column number to obtain the column’s value is more efficient than using the column’s name. The column number is similar to an array subscript in that the column number provides direct access to the specified column. Using the column’s name requires a linear search of the column names to locate the appropriate column.

For simplicity, this example treats each column’s value as an `Object`. The program retrieves each column value with **ResultSet** method `getObject` (line 54) and appends the `String` representation of the `Object` to `results`. When **ResultSet** processing completes, the program closes the **Statement** (line 63) and the database **Connection** (line 64). Notice that, unlike array subscripts, which start at 0, **ResultSet** column numbers start at 1.

**Common Programming Error 8.9**

Specifying column number 0 when obtaining values from a **ResultSet** causes an `SQLException`.

**Common Programming Error 8.10**

Attempting to manipulate a **ResultSet** after closing the **Statement** that created the **ResultSet** causes an `SQLException`. The program discards the **ResultSet** when the corresponding **Statement** is closed.

**Software Engineering Observation 8.8**

Each **Statement** object can open only one **ResultSet** object at a time. When a **Statement** returns a new **ResultSet**, the **Statement** closes the prior **ResultSet**. To use multiple **ResultSets** in parallel, separate **Statement** objects must return the **ResultSets**.

Lines 66–73 create the GUI that displays the `StringBuffer results`, set the size of the application window and show the application window.

To run this example as well as the others in the Chapter, the `classpath` in the following command line must be used

```
java -classpath f:\Cloudscape_3.6\lib\cloudscape.jar;
f:\Cloudscape_3.6\frameworks\RmiJdbc\classes\RmiJdbc.jar;.
com.deitel.advjhtp1.jdbc.DisplayAuthors
```

1. You will need to modify this command line to indicate the proper location of your Cloudscape installation and to use directory separators that are appropriate to your operating system (e.g., colons on UNIX/Linux).
These JARs (cloudscape.jar and RmiJdbc.jar) must be present in the classpath or none of the examples in this Chapter will execute. Note that the classpath includes . for the current directory. When setting the classpath at the command line, this ensures that the interpreter can locate classes in the current directory. You can also set the classpath using Cloudscape’s setServerCloudscapeCP batch file or shell script discussed earlier in this section.

### 8.6.2 Querying the books Database

The next example (Fig. 8.28 and Fig. 8.31) enhances the example of Fig. 8.26 by allowing the user to enter any query into the program. The example displays the results of a query in a JTable, using aTableModel object to provide the ResultSet data to the JTable. Class ResultSetTableModel (Fig. 8.28) performs the connection to the database and maintains the model. Class DisplayQueryResults (Fig. 8.31) creates the GUI and specifies an instance of class ResultSetTableModel as the model for the JTable.

Class ResultSetTableModel (Fig. 8.28) extends class AbstractTableModel (package javax.swing.table), which implements interface TableModel. Class ResultSetTableModel overridesTableModel methods getColumnClass, getColumnCount, getColumnName, getRowCount and getValueAt. The default implementations of TableModel methods isCellEditable and setValueAt (provided by AbstractTableModel) are not overridden, because this example does not support editing the JTable cells. Also, the default implementations of TableModel methods addTableModelListener and removeTableModelListener (provided by AbstractTableModel) are not overridden, because the implementations of these methods from AbstractTableModel properly add and remove event listeners.

```java
// Fig. 8.28: ResultSetTableModel.java
// A TableModel that supplies ResultSet data to a JTable.
package com.deitel.advjhtp1.jdbc;

// Java core packages
import java.sql.*;
import java.util.*;

// Java extension packages
import javax.swing.table.*;

// ResultSet rows and columns are counted from 1 and JTable
// rows and columns are counted from 0. When processing
// ResultSet rows or columns for use in a JTable, it is
// necessary to add 1 to the row or column number to manipulate
// the appropriate ResultSet column (i.e., JTable column 0 is
// ResultSet column 1 and JTable row 0 is ResultSet row 1).
public class ResultSetTableModel extends AbstractTableModel {
    private Connection connection;
    private Statement statement;
}
```

Fig. 8.28 ResultSetTableModel enables a JTable to display the contents of a ResultSet (part 1 of 4).
private ResultSet resultSet;
private ResultSetMetaData metaData;
private int numberOfRows;

// initialize resultSet and obtain its meta data object;
// determine number of rows
public ResultSetTableModel( String driver, String url, String query ) throws SQLException, ClassNotFoundException {
    // load database driver class
    Class.forName( driver );

    // connect to database
    connection = DriverManager.getConnection( url );

    // create Statement to query database
    statement = connection.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY );

    // set query and execute it
    setQuery( query );
}

// get class that represents column type
public Class getColumnClass( int column )
{
    // determine Java class of column
    try {
        String className =
            metaData.getColumnClassName( column + 1 );

        // return Class object that represents className
        return Class.forName( className );
    }

    // catch SQLExceptions and ClassNotFoundExceptions
    catch ( Exception exception ) {
        exception.printStackTrace();
    }

    // if problems occur above, assume type Object
    return Object.class;
}

// get number of columns in ResultSet
public int getColumnCount()
{
    // determine number of columns
    try {
        return metaData.getColumnCount();
    }
}

Fig. 8.28 ResultSetTableModel enables a JTable to display the contents of a ResultSet (part 2 of 4).
```java
// catch SQLExceptions and print error message
try {
    resultSet.getObject( column + 1 );
} catch ( SQLException sqlException ) {
    sqlException.printStackTrace();
}

// if problems, return empty string object
return "";
```
The `ResultSetTableModel` constructor (lines 27–43) receives three `String` arguments—the driver class name, the URL of the database and the default query to perform. The constructor throws any exceptions that occur in its body back to the application that created the `ResultSetTableModel` object, so that the application can determine how to handle the exception (e.g., report an error and terminate the application). Line 31 loads the database driver. Line 34 establishes a connection to the database. Line 37 invokes `Connection` method `createStatement` to create a `Statement` object. This example uses a version of method `createStatement` that takes two arguments—the `result-set type` and the `result-set concurrency`. The result-set type (Fig. 8.29) specifies whether the `ResultSet`’s cursor is able to scroll in both directions or forward only and whether the `ResultSet` is sensitive to changes. `ResultSet`s that are sensitive to changes reflect those changes immediately after they are made with methods of interface `ResultSet`. If a `ResultSet` is insensitive to changes, the query that produced the `ResultSet` must be executed again to reflect any changes made. The result-set concurrency (Fig. 8.30) specifies whether the `ResultSet` can be updated with `ResultSet`’s update methods. This example uses a `ResultSet` that is scrollable, insensitive to changes.
and read only. Line 42 invokes `ResultSetTableModel` method `setQuery` (lines 141–155) to perform the default query.

**Portability Tip 8.6**

Some `ResultSet` implementations do not support scrollable `ResultSet`. In such cases, typically the driver returns a `ResultSet` in which the cursor can move only forward. For more information, see your database-driver documentation.

**Portability Tip 8.7**

Some `ResultSet` implementations do not support updatable `ResultSet`. In such cases, typically the driver returns a read-only `ResultSet`. For more information, see your database-driver documentation.

**Common Programming Error 8.11**

Attempting to update a `ResultSet` when the database driver does not support updatable `ResultSets` causes `SQLException`.

**Common Programming Error 8.12**

Attempting to move the cursor backwards through a `ResultSet` when the database driver does not support backwards scrolling causes an `SQLException`.

Method `getColumnClass` (lines 46–64) returns a `Class` object that represents the superclass of all objects in a particular column. The `JTable` uses this information to configure the default cell renderer and cell editor for that column in the `JTable`. Lines 50–51 use `ResultSetMetaData` method `getColumnClassName` to obtain the fully qualified class name for the specified column. Line 54 loads the class definition for that class and returns the corresponding `Class` object. If an exception occurs, the `catch` at lines 58–60 prints a stack trace and line 63 returns `Object.class`—the `Class` instance that represents class `Object`—as the default type. [Note: Line 51 uses the argument `column + 1`. Like arrays, `JTable` row and column numbers are counted from 0. However, `ResultSet` row and column numbers are counted from 1. Thus, when processing `ResultSet` rows or columns for use in a `JTable`, it is necessary to add 1 to the row or column number to manipulate the appropriate `ResultSet` row or column.]

<table>
<thead>
<tr>
<th><code>ResultSet</code> static type constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TYPE_FORWARD_ONLY</code></td>
<td>Specifies that a <code>ResultSet</code>’s cursor can move only in the forward direction (i.e., from the first record to the last record in the <code>ResultSet</code>).</td>
</tr>
<tr>
<td><code>TYPE_SCROLL_INSENSITIVE</code></td>
<td>Specifies that a <code>ResultSet</code>'s cursor can scroll in either direction and that the changes made to the <code>ResultSet</code> during <code>ResultSet</code> processing are not reflected in the <code>ResultSet</code> unless the program queries the database again.</td>
</tr>
</tbody>
</table>

![Fig. 8.29](image)  
**Fig. 8.29** `ResultSet` constants for specifying `ResultSet` type (part 1 of 2).
Method `getColumnCount` (lines 67–81) returns the number of columns in the model’s underlying `ResultSet`. Line 71 uses `ResultSetMetaData` method `getColumnCount` to obtain the number of columns in the `ResultSet`. If an exception occurs, the `catch` at lines 75–77 prints a stack trace and line 80 returns 0 as the default number of columns.

Method `getColumnName` (lines 84–98) returns the name of the column in the model’s underlying `ResultSet`. Line 88 uses `ResultSetMetaData` method `getColumnName` to obtain the column name from the `ResultSet`. If an exception occurs, the `catch` at lines 92–94 prints a stack trace and line 97 returns the empty string as the default column name.

Method `getRowCount` (lines 101–104) returns the number of rows in the model’s underlying `ResultSet`. When method `setQuery` (lines 141–155) performs a query, it stores the number of rows in variable `numberOfRows`.

Method `getValueAt` (lines 107–123) returns the `Object` in a particular row and column of the model’s underlying `ResultSet`. Line 111 uses `ResultSet` method `absolute` to position the `ResultSet` cursor at a specific row. Line 113 uses `ResultSet` method `getObject` to obtain the `Object` in a specific column of the current row. If an exception occurs, the `catch` at lines 117–119 prints a stack trace and line 122 returns the empty string object as the default value.

Method `finalize` (lines 126–138) closes the `Statement` and `Connection` objects if a `ResultSetTableModel` object is garbage collected.
Method `setQuery` (lines 141–155) executes the query it receives as an argument to obtain a new `ResultSet` (line 144). Line 147 gets the `ResultSetMetaData` for the new `ResultSet`. Line 150 uses `ResultSet` method `last` to position the `ResultSet` cursor at the last row in the `ResultSet`. Line 151 uses `ResultSet` method `getRow` to obtain the row number for the current row in the `ResultSet`. Line 154 invokes method `fireTableStructureChanged` (inherited from class `AbstractTableModel`) to notify any `JTable` using this `ResultSetTableModel` object as its model that the structure of the model has changed (i.e., the underlying `ResultSet` contains new data or new columns). This causes the `JTable` to repopulate its rows and columns with the new `ResultSet` data. Method `setQuery` throws any exceptions that occur in its body back to the application that invoked `setQuery`.

The `DisplayQueryResults` (Fig. 8.31) constructor (lines 21–121) creates a `ResultSetTableModel` object and defines the GUI for the application. Lines 26, 29 and 32 define the database driver class name, database URL and default query that are passed to the `ResultSetTableModel` constructor to make the initial connection to the database and perform the default query. Line 61 creates the `JTable` object and passes a `ResultSetTableModel` object to the `JTable` constructor, which then registers the `JTable` as a listener for `TableModelEvents` generated by the `ResultSetTableModel`. Lines 70–94 register an event handler for the `submitButton` that the user clicks to submit a query to the database. When the user clicks the button, method `actionPerformed` (lines 75–90) invokes `ResultSetTableModel` method `setQuery` to execute the new query. The screen captures in Fig. 8.31 show the results of two queries. The first screen capture shows the default query that selects all the authors from table `authors` of database `books`. The second screen capture shows a query that selects each author’s first name and last name from the `authors` table and combines that information with the title and edition number from the `titles` table. Try entering your own queries in the text area and clicking the `Submit Query` button to execute the query.

```java
// Fig. 8.31: DisplayQueryResults.java
// Display the contents of the Authors table in the
// Books database.
package com.deitel.advjhtp1.jdbc;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.util.*;

// Java extension packages
import javax.swing.*;
import javax.swing.table.*;

public class DisplayQueryResults extends JFrame {
    private ResultSetTableModel tableModel;
    private JTextArea queryArea;
```
// create ResultSetTableModel and GUI
public DisplayQueryResults()
{
    super( "Displaying Query Results" );

    // Cloudscape database driver class name
    String driver = "COM.cloudscape.core.RmiJdbcDriver";

    // URL to connect to books database
    String url = "jdbc:cloudscape:rmi:books";

    // query to select entire authors table
    String query = "SELECT * FROM authors";

    // create ResultSetTableModel and display database table
    try {
        // create TableModel for results of query
        // SELECT * FROM authors
        tableModel =
            new ResultSetTableModel( driver, url, query );

        // set up JTextArea in which user types queries
        queryArea = new JTextArea( query, 3, 100 );
        queryArea.setWrapStyleWord( true );
        queryArea.setLineWrap( true );

        JScrollPane scrollPane = new JScrollPane( queryArea,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );

        // set up JButton for submitting queries
        JButton submitButton = new JButton( "Submit Query" );

        // create Box to manage placement of queryArea and
        // submitButton in GUI
        Box box = Box.createHorizontalBox();
        box.add( scrollPane );
        box.add( submitButton );

        // create JTable delegate for tableModel
        JTable resultTable = new JTable( tableModel );

        // place GUI components on content pane
        Container c = getContentPane();
        c.add( box, BorderLayout.NORTH );
        c.add( new JScrollPane( resultTable ),
            BorderLayout.CENTER );

        // create event listener for submitButton
        submitButton.addActionListener(new ActionListener() {
        }}

Fig. 8.31 DisplayQueryResults enables a user to query database books.
public void actionPerformed(ActionEvent e) {
    try {
        tableModel.setQuery(queryArea.getText());
    }
    catch (SQLException sqlException) {
        JOptionPane.showMessageDialog(null, sqlException.toString(),
                               "Database error", JOptionPane.ERROR_MESSAGE);
    }
}

// set window size and display window
setSize(500, 250);
setVisible(true);

// set window size and display window
setSize(500, 250);
setVisible(true);

System.exit(1); // terminate application

// catch SQLException thrown by ResultSetTableModel if database driver not found
catch (ClassNotFoundException classNotFound) {
    JOptionPane.showMessageDialog(null, "Cloudscape driver not found", "Driver not found",
                                   JOptionPane.ERROR_MESSAGE);
    System.exit(1); // terminate application
}

// catch SQLException thrown by ResultSetTableModel if database driver not found
// if problems occur while setting up database
// connection and querying database
catch (SQLException sqlException) {
    JOptionPane.showMessageDialog(null, sqlException.toString(),
                                   "Database error", JOptionPane.ERROR_MESSAGE);
    System.exit(1); // terminate application
}

// execute application
public static void main(String args[]) {
    Fig. 8.31 DisplayQueryResults enables a user to query database books.
8.7 Case Study: Address-Book Application

Our next example uses the SQL and JDBC concepts presented so far to implement a substantial address-book application that enables the user to insert, locate, update and delete address-book entries in the Cloudscape database `addressbook`. [Note: An SQL script to create this database is provided with the example code for this chapter. Section 8.5 demonstrates executing an SQL script with Cloudscape.]

Database `addressbook` contains four tables: `names`, `addresses`, `phoneNumbers` and `emailAddresses`. Figure 8.32 shows the relationships between the tables. The first line in each table is the table’s name. Each table’s primary-key field is highlighted in green. Tables `addresses`, `phoneNumbers` and `emailAddresses` each have `personID` as a foreign key. Thus, a program cannot place records in those tables unless the `personID` is a valid value in table `names`. Although the address-book application currently allows only one address, one phone number and one e-mail address per person, the database was designed to support multiple addresses, phone numbers and e-mail addresses for each person. So there is a one-to-many relationship between a record in the `names` table and records in the other tables. Note that the relationship lines between the tables link the foreign key (`personID`) of tables `phoneNumbers`, `emailAddresses` and `addresses` to the primary key of table `names`. In each of the tables, fields `personID`, `addressID`, `emailID` and `phoneID` are integers. All other fields are strings.
This example introduces two new concepts: `PreparedStatement` and `transaction processing`. Section 8.7.1 and Section 8.7.2 discuss these new concepts. Then, Section 8.7.3 presents the `AddressBook` application and its supporting classes.

### 8.7.1 PreparedStatements

Interface `PreparedStatement` enables an application programmer to create SQL statements that are maintained in a compiled form that enables the statements to execute more efficiently than `Statement` objects. `PreparedStatement` objects also are more flexible than `Statement` objects, because they can specify parameters. This allows programs to execute the same query repeatedly with different parameter values. For example, in the `books` database, you might want to locate all book titles for an author with a specific last name and first name, and you might want to execute that query for several authors. With a `PreparedStatement`, that query is defined as follows:

```java
PreparedStatement authorBooks = connection.prepareStatement(
    "SELECT lastName, firstName, title " +
    "FROM authors, titles, authorISBN " +
    "WHERE authors.authorID = authorISBN.authorID AND " +
    " titles.ISBN = authorISBN.isbn AND " +
    " lastName = ? AND firstName = ?"
);  
```

Note the two `question marks` (?) in the last line of the preceding statement. These characters represent placeholders for values that will be passed as part of the query to the database. Before the program executes a `PreparedStatement`, the program must specify the values of those parameters by using the `set` methods of interface `PreparedStatement`. 
For the preceding query, both parameters are strings that can be set with `PreparedStatement` method `setString` as follows:

```java
authorBooks.setString( 1, "Deitel" );
authorBooks.setString( 2, "Paul" );
```

Method `setString`'s first argument represents the number of the parameter being set and the second argument is the value to set for that parameter. Parameter numbers are counted from 1, starting with the first question mark (\?) in the SQL statement. When the program executes the preceding `PreparedStatement` with the parameter values shown here, the SQL statement passed to the database is

```sql
SELECT lastName, firstName, title
FROM authors, titles, authorISBN
WHERE authors.authorID = authorISBN.authorID AND
      titles.ISBN = authorISBN.isbn AND
      lastName = 'Deitel' AND firstName = 'Paul'
```

It is important to note that method `setString` escapes `String` parameter values properly. For example, if the last name is O’Brien, the statement

```java
authorBooks.setString( 1, "O'Brien" );
```

escapes the \' character in O’Brien by replacing it with two single-quote characters.

### Performance Tip 8.4

In programs that execute SQL statements multiple times with different parameter values, `PreparedStatements` are more efficient than `Statements`, because `PreparedStatements` maintain the SQL statement in a compiled format. This is a very important performance enhancement.

### Software Engineering Observation 8.9

`PreparedStatements` are more flexible than `Statements`, because `PreparedStatements` support customization of a query with parameter values. With a `Statement`, the program must create a new `String` containing an SQL statement for each new query.

### Good Programming Practice 8.2

Use `PreparedStatements` with parameters for queries that receive `String` values as arguments to ensure that the `Strings` are quoted properly in the SQL statement.

Interface `PreparedStatement` provides `set` methods for each SQL type supported. It is important to use the `set` method that is appropriate for the SQL type of the parameter in the database; otherwise, `SQLExceptions` can occur when the program attempts to convert the parameter value to an incorrect type. For a complete list of these `set` methods, see the Java API documentation for interface `PreparedStatement`.

### Common Programming Error 8.13

Using the incorrect `PreparedStatement` set method can cause `SQLExceptions` if an attempt is made to convert a parameter value to an incorrect data type.
8.7.2 Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the applications continue processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination account. Next, you specify the destination account. The bank checks the source account to determine if there are sufficient funds in the account to complete the transfer. If so, the bank withdraws the specified amount from the source account and, if all goes well, deposits the money into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. How would you feel if the money was subtracted from your source account and the bank did not deposit the money in the destination account?

Transaction processing enables a program that interacts with a database to treat a database operation (or set of operations) as a single operation. Such an operation also is known as an atomic operation or a transaction. At the end of a transaction, a decision can be made either to commit the transaction or back the transaction. Committing the transaction finalizes the database operation(s); all insertions, updates and deletions performed as part of the transaction cannot be reversed without performing a new database operation. Rolling back the transaction leaves the database in its state prior to the database operation. This is useful when a portion of a transaction fails to complete properly. In our bank-account-transfer discussion, the transaction would be rolled back if the deposit could not be made into the destination account.

Java provides transaction processing via methods of interface Connection. Method setAutoCommit specifies whether each SQL statement commits after it completes (a true argument) or if several SQL statements should be grouped as a transaction (a false argument). If the argument to setAutoCommit is false, the program must follow the last SQL statement in the transaction with a call to Connection method commit (to commit the changes to the database) or Connection method rollback (to return the database to its state prior to the transaction). Interface Connection also provides method getAutoCommit to determine the autocommit state for the Connection.

Software Engineering Observation 8.10
By default, a Connection is in autocommit mode.

Software Engineering Observation 8.11
Most JDBC drivers support transaction processing. Those that do not are not JDBC compliant drivers.

8.7.3 Address-Book Application

The address-book application consists of five classes and interfaces: class AddressBookEntry (Fig. 8.33), interface AddressBookDataAccess (Fig. 8.34), class DataAccessException (Fig. 8.35), class CloudscapeDataAccess (Fig. 8.36), class AddressBookEntryFrame (Fig. 8.37) and class AddressBook (Fig. 8.38).
Class **AddressBookEntry** (Fig. 8.33) represents the data for an entry in the address book. The class contains properties for all the fields in the four tables of database `addressbook`.

```java
// Fig. 8.33: AddressBookEntry.java
// JavaBean to represent one address book entry.
package com.deitel.advjhtp1.jdbc.addressbook;

public class AddressBookEntry {
    private String firstName = "";
    private String lastName = "";
    private String address1 = "";
    private String address2 = "";
    private String city = "";
    private String state = "";
    private String zipcode = "";
    private String phoneNumber = "";
    private String emailAddress = "";
    private int personID;
    private int addressID;
    private int phoneID;
    private int emailID;

    // empty constructor
    public AddressBookEntry()
    {
    }

    // set person's id
    public AddressBookEntry( int id )
    {
        personID = id;
    }

    // set person's first name
    public void setFirstName( String first )
    {
        firstName = first;
    }

    // get person's first name
    public String getFirstName()
    {
        return firstName;
    }

    // set person's last name
    public void setLastName( String last )
    {
        lastName = last;
    }
}
```

**Fig. 8.33**  *AddressBookEntry* bean represents an address book entry (part 1 of 4).
// get person's last name
public String getLastName()
{
    return lastName;
}

// set first line of person's address
public void setAddress1( String firstLine )
{
    address1 = firstLine;
}

// get first line of person's address
public String getAddress1()
{
    return address1;
}

// set second line of person's address
public void setAddress2( String secondLine )
{
    address2 = secondLine;
}

// get second line of person's address
public String getAddress2()
{
    return address2;
}

// set city in which person lives
public void setCity( String personCity )
{
    city = personCity;
}

// get city in which person lives
public String getCity()
{
    return city;
}

// set state in which person lives
public void setState( String personState )
{
    state = personState;
}

// get state in which person lives
public String getState()
{
    return state;
}

AddressBookEntry bean represents an address book entry (part 2 of 4).
// set person's zip code
public void setZipcode( String zip )
{
    zipcode = zip;
}

// get person's zip code
public String getZipcode()
{
    return zipcode;
}

// set person's phone number
public void setPhoneNumber( String number )
{
    phoneNumber = number;
}

// get person's phone number
public String getPhoneNumber()
{
    return phoneNumber;
}

// set person's email address
public void setEmailAddress( String email )
{
    emailAddress = email;
}

// get person's email address
public String getEmailAddress()
{
    return emailAddress;
}

// get person's ID
public int getPersonID()
{
    return personID;
}

// set person's addressID
public void setAddressID( int id )
{
    addressID = id;
}

// get person's addressID
public int getAddressID()
{
    return addressID;
}

Fig. 8.33  AddressBookEntry  bean represents an address book entry (part 3 of 4).
Fig. 8.34 AddressBookDataAccess interface describes the methods for accessing the addressbook database (part 1 of 2).

Fig. 8.33 AddressBookEntry bean represents an address book entry (part 4 of 4).

Interface AddressBookDataAccess (Fig. 8.34) describes methods required by the address-book application to perform insertions, updates, deletions and searches with the addressbook database. Any class that implements this interface can be used by the AddressBook application class to interact with the database. Thus, if you want to modify the application to use a database other than Cloudscape, you can do so by providing your own implementation of class AddressBookDataAccess.

```java
// Fig. 8.34: AddressBookDataAccess.java
// Interface that specifies the methods for inserting, updating, deleting and finding records.
package com.deitel.advjhtp1.jdbc.addressbook;

import java.sql.*;

public interface AddressBookDataAccess {

    // Locate specified person by last name. Return AddressBookEntry containing information.
    public AddressBookEntry findPerson( String lastName );
}
```

```java
// Fig. 8.34: AddressBookDataAccess.java
// Interface that specifies the methods for inserting, updating, deleting and finding records.
package com.deitel.advjhtp1.jdbc.addressbook;

import java.sql.*;

public interface AddressBookDataAccess {

    // Locate specified person by last name. Return AddressBookEntry containing information.
    public AddressBookEntry findPerson( String lastName );
}
```
Interface `AddressBookDataAccess` contains five methods. Method `findPerson` (line 13) receives an `String` argument containing the last name of the person for which to search. The method returns the `AddressBookEntry` containing the person’s complete information if the person was found in the database; otherwise, the method returns `false`. Method `savePerson` (lines 17–18) receives an `AddressBookEntry` argument containing the data to save and updates the corresponding record in the database. Method `newPerson` (lines 22–23) receives an `AddressBookEntry` argument containing the information for a new person and inserts the person’s information in the database. Method `deletePerson` (lines 27–28) receives an `AddressBookEntry` argument containing the person to delete from the database and uses the `personID` to remove the records that represent the person from all four tables in the database. Method `close` (line 31) closes the statements and the connection to the database.

Class `DataAccessException` (Fig. 8.35) extends class `Exception`. Some of the methods of interface `AddressBookDataAccess` throw `DataAccessException` when there is a problem with the data source connection.
Fig. 8.35 **DataAccessException** is thrown when there is a problem accessing the data source (part 2 of 2).

Class **CloudscapeDataAccess** (Fig. 8.36) implements interface **AddressBookDataAccess** to interact with our **addressbook** database in Cloudscape. An object of this class contains a reference to a **Connection** object (line 13) that maintains the connection to the database and several references to **PreparedStatement** objects (lines 16–37) that represent the interactions with the database for inserting, updating, deleting and finding records.

```java
// Fig. 8.36: CloudscapeDataAccess.java
// An implementation of interface AddressBookDataAccess that
// performs database operations with PreparedStatements.
package com.deitel.advjhtp1.jdbc.addressbook;

// Java core packages
import java.sql.*;

public class CloudscapeDataAccess
    implements AddressBookDataAccess {
    // reference to database connection
    private Connection connection;
    // reference to prepared statement for locating entry
    private PreparedStatement sqlFind;
    // reference to prepared statement for determining personID
    private PreparedStatement sqlPersonID;
```

Fig. 8.36 **CloudscapeDataAccess** implements interface **AddressBookDataAccess** to perform the connection to the database and the database interactions (part 1 of 10).
public CloudscapeDataAccess() throws Exception
{
    // connect to addressbook database
    connect();

    // locate person
    sqlFind = connection.prepareStatement(
        "SELECT names.personID, firstName, lastName, " +
        "addressID, address1, address2, city, state, " +
        "zipcode, phoneID, phoneNumber, emailID, " +
        "emailAddress " +
        "FROM names, addresses, phoneNumbers, emailAddresses " +
        "WHERE lastName = ? AND " +
        "names.personID = addresses.personID AND " +
        "names.personID = phoneNumbers.personID AND " +
        "names.personID = emailAddresses.personID" );

    // Obtain personID for last person inserted in database.
    // [This is a Cloudscape-specific database operation.]
    sqlPersonID = connection.prepareStatement(
        "VALUES ConnectionInfo.lastAutoincrementValue( " +
        "'APP', 'NAMES', 'PERSONID')" );

    // Insert first and last names in table names.
    // For referential integrity, this must be performed
    // before sqlInsertAddress, sqlInsertPhone and
    // sqlInsertEmail.
    sqlInsertName = connection.prepareStatement(
        "INSERT INTO names ( firstName, lastName ) " +
        "VALUES (?, ?)" );
// insert address in table addresses
sqlInsertAddress = connection.prepareStatement(
    "INSERT INTO addresses ( personID, address1, " +
    "address2, city, state, zipcode ) " +
    "VALUES ( ?, ?, ?, ?, ?, ? )" );

// insert phone number in table phoneNumbers
sqlInsertPhone = connection.prepareStatement(
    "INSERT INTO phoneNumbers " +
    "( personID, phoneNumber ) " +
    "VALUES ( ?, ? )" );

// insert email in table emailAddresses
sqlInsertEmail = connection.prepareStatement(
    "INSERT INTO emailAddresses " +
    "( personID, emailAddress ) " +
    "VALUES ( ?, ? )" );

// update first and last names in table names
sqlUpdateName = connection.prepareStatement(
    "UPDATE names SET firstName = ?, lastName = ? " +
    "WHERE personID = ?" );

// update address in table addresses
sqlUpdateAddress = connection.prepareStatement(
    "UPDATE addresses SET address1 = ?, address2 = ?, " +
    "city = ?, state = ?, zipcode = ? " +
    "WHERE addressID = ?" );

// update phone number in table phoneNumbers
sqlUpdatePhone = connection.prepareStatement(
    "UPDATE phoneNumbers SET phoneNumber = ? " +
    "WHERE phoneID = ?" );

// update email in table emailAddresses
sqlUpdateEmail = connection.prepareStatement(
    "UPDATE emailAddresses SET emailAddress = ? " +
    "WHERE emailID = ?" );

// Delete row from table names. This must be executed
// after sqlDeleteAddress, sqlDeletePhone and
// sqlDeleteEmail, because of referential integrity.
sqlDeleteName = connection.prepareStatement(
    "DELETE FROM names WHERE personID = ?" );

// delete address from table addresses
sqlDeleteAddress = connection.prepareStatement(
    "DELETE FROM addresses WHERE personID = ?" );

Fig. 8.36  CloudscapeDataAccess implements interface
AddressBookDataAccess to perform the connection to the
database and the database interactions (part 3 of 10).
// delete phone number from table phoneNumbers
sqlDeletePhone = connection.prepareStatement(
    "DELETE FROM phoneNumbers WHERE personID = ?" );

// delete email address from table emailAddresses
sqlDeleteEmail = connection.prepareStatement(
    "DELETE FROM emailAddresses WHERE personID = ?" );

}  // end CloudscapeDataAccess constructor

// Obtain a connection to addressbook database. Method may
// may throw ClassNotFoundException or SQLException. If so,
// exception is passed via this class's constructor back to
// the AddressBook application so the application can display
// an error message and terminate.
private void connect() throws Exception {
    // Cloudscape database driver class name
    String driver = "COM.cloudscape.core.RmiJdbcDriver";

    // URL to connect to addressbook database
    String url = "jdbc:cloudscape:rmi:addressbook";

    // load database driver class
    Class.forName( driver );

    // connect to database
    connection = DriverManager.getConnection( url );

    // Require manual commit for transactions. This enables
    // the program to rollback transactions that do not
    // complete and commit transactions that complete properly.
    connection.setAutoCommit( false );
}

// Locate specified person. Method returns AddressBookEntry
// containing information.
public AddressBookEntry findPerson( String lastName ) {
    try {
        // set query parameter and execute query
        sqlFind.setString( 1, lastName );
        ResultSet resultSet = sqlFind.executeQuery();

        // if no records found, return immediately
        if ( !resultSet.next() )
            return null;

        // create new AddressBookEntry
        AddressBookEntry person = new AddressBookEntry(
            resultSet.getInt( 1 ) );
    }

    CloudscapeDataAccess implements interface
    AddressBookDataAccess to perform the connection to the
database and the database interactions (part 4 of 10).
// set AddressBookEntry properties
person.setFirstName( resultSet.getString( 2 ) );
person.setLastName( resultSet.getString( 3 ) );

person.setAddressID( resultSet.getInt( 4 ) );
person.setAddress1( resultSet.getString( 5 ) );
person.setAddress2( resultSet.getString( 6 ) );
person.setCity( resultSet.getString( 7 ) );
person.setState( resultSet.getString( 8 ) );
person.setZipcode( resultSet.getString( 9 ) );

person.setPhoneID( resultSet.getInt( 10 ) );
person.setPhoneNumber( resultSet.getString( 11 ) );

person.setEmailID( resultSet.getInt( 12 ) );
person.setEmailAddress( resultSet.getString( 13 ) );

// return AddressBookEntry
return person;
}

// catch SQLException
try ( SQLException sqlException ) {
    return null;
}

} // end method findPerson

// Update an entry. Method returns boolean indicating
// success or failure.
public boolean savePerson( AddressBookEntry person )
    throws DataAccessException
{
    // update person in database
    try {
        int result;

        // update names table
        sqlUpdateName.setString( 1, person.getFirstName() );
        sqlUpdateName.setString( 2, person.getLastName() );
        sqlUpdateName.setInt( 3, person.getPersonID() );
        result = sqlUpdateName.executeUpdate();

        // if update fails, rollback and discontinue
        if ( result == 0 ) {
            connection.rollback(); // rollback update
            return false; // update unsuccessful
        }
    }

    // update addresses table
    sqlUpdateAddress.setString( 1, person.getAddress1() );
    sqlUpdateAddress.setString( 2, person.getAddress2() );

Fig. 8.36 CloudscapeDataAccess implements interface
AddressBookDataAccess to perform the connection to the
database and the database interactions (part 5 of 10).
sqlUpdateAddress.setString(3, person.getCity());
sqlUpdateAddress.setString(4, person.getState());
sqlUpdateAddress.setString(5, person.getZipcode());
sqlUpdateAddress.setInt(6, person.getAddressID());
result = sqlUpdateAddress.executeUpdate();

// if update fails, rollback and discontinue
if (result == 0) {
    connection.rollback(); // rollback update
    return false; // update unsuccessful
}

// update phoneNumbers table
sqlUpdatePhone.setString(1, person.getPhoneNumber());
sqlUpdatePhone.setInt(2, person.getPhoneID());
result = sqlUpdatePhone.executeUpdate();

// if update fails, rollback and discontinue
if (result == 0) {
    connection.rollback(); // rollback update
    return false; // update unsuccessful
}

// update emailAddresses table
sqlUpdateEmail.setString(1, person.getEmailAddress());
sqlUpdateEmail.setInt(2, person.getEmailID());
result = sqlUpdateEmail.executeUpdate();

// if update fails, rollback and discontinue
if (result == 0) {
    connection.rollback(); // rollback update
    return false; // update unsuccessful
}

 connection.commit(); // commit update
return true; // update successful

} // end try

// detect problems updating database
catch (SQLException sqlException) {

    // rollback transaction
    try {
        connection.rollback(); // rollback update
        return false; // update unsuccessful
    }

    // handle exception rolling back transaction
    catch (SQLException exception) {
        throw new DataAccessException(exception);
    }

Fig. 8.36 **CloudscapeDataAccess** implements interface **AddressBookDataAccess** to perform the connection to the database and the database interactions (part 6 of 10).
private static Connection connection = null;

Fig. 8.36 CloudscapeDataAccess implements interface AddressBookDataAccess to perform the connection to the database and the database interactions (part 7 of 10).
sqlInsertPhone.setString( 2,
    person.getPhoneNumber() );
result = sqlInsertPhone.executeUpdate();

// if insert fails, rollback and discontinue
if ( result == 0 ) {
    connection.rollback(); // rollback insert
    return false;          // insert unsuccessful
}

// insert email address in emailAddresses table
sqlInsertEmail.setInt( 1, personID );
sqlInsertEmail.setString( 2,
    person.getEmailAddress() );
result = sqlInsertEmail.executeUpdate();

// if insert fails, rollback and discontinue
if ( result == 0 ) {
    connection.rollback(); // rollback insert
    return false;          // insert unsuccessful
}

connection.commit(); // commit insert
return true;         // insert successful
}
else
    return false;
}  // end try

// detect problems updating database
catch ( SQLException sqlException ) {
    // rollback transaction
    try {
        connection.rollback(); // rollback update
        return false;          // update unsuccessful
    }

    // handle exception rolling back transaction
    catch ( SQLException exception ) {
        throw new DataAccessException( exception );
    }
}
}  // end method newPerson

// Delete an entry. Method returns boolean indicating
// success or failure.
public boolean deletePerson( AddressBookEntry person )
    throws DataAccessException
    { Fig. 8.36  CloudscapeDataAccess implements interface
AddressBookDataAccess to perform the connection to the
database and the database interactions (part 8 of 10).
// delete a person from database
try {
    int result;

    // delete address from addresses table
    sqlDeleteAddress.setInt(1, person.getPersonID());
    result = sqlDeleteAddress.executeUpdate();

    // if delete fails, rollback and discontinue
    if (result == 0) {
        connection.rollback(); // rollback delete
        return false; // delete unsuccessful
    }

    // delete phone number from phoneNumbers table
    sqlDeletePhone.setInt(1, person.getPersonID());
    result = sqlDeletePhone.executeUpdate();

    // if delete fails, rollback and discontinue
    if (result == 0) {
        connection.rollback(); // rollback delete
        return false; // delete unsuccessful
    }

    // delete email address from emailAddresses table
    sqlDeleteEmail.setInt(1, person.getPersonID());
    result = sqlDeleteEmail.executeUpdate();

    // if delete fails, rollback and discontinue
    if (result == 0) {
        connection.rollback(); // rollback delete
        return false; // delete unsuccessful
    }

    // delete name from names table
    sqlDeleteName.setInt(1, person.getPersonID());
    result = sqlDeleteName.executeUpdate();

    // if delete fails, rollback and discontinue
    if (result == 0) {
        connection.rollback(); // rollback delete
        return false; // delete unsuccessful
    }

    connection.commit(); // commit delete
    return true; // delete successful
} // end try

// detect problems updating database
catch (SQLException sqlException) {
    // rollback transaction
}

Fig. 8.36 CloudscapeDataAccess implements interface AddressBookDataAccess to perform the connection to the database and the database interactions (part 9 of 10).
try {
    connection.rollback(); // rollback update
    return false;          // update unsuccessful
}

// handle exception rolling back transaction
catch (SQLException exception) {
    throw new DataAccessException( exception );
}

// method to close statements and database connection
public void close() {
    try {
        sqlFind.close();
        sqlPersonID.close();
        sqlInsertName.close();
        sqlInsertAddress.close();
        sqlInsertPhone.close();
        sqlInsertEmail.close();
        sqlUpdateName.close();
        sqlUpdateAddress.close();
        sqlUpdatePhone.close();
        sqlUpdateEmail.close();
        sqlDeleteName.close();
        sqlDeleteAddress.close();
        sqlDeletePhone.close();
        sqlDeleteEmail.close();
        connection.close();
    } // end try

    // detect problems closing statements and connection
    catch (SQLException sqlException) {
        sqlException.printStackTrace();
    }
}  // end method close

// Method to clean up database connection. Provided in case
// CloudscapeDataAccess object is garbage collected.
protected void finalize() {
    close();
}
}  // end class CloudscapeDataAccess

Fig. 8.36 CloudscapeDataAccess implements interface AddressBookDataAccess to perform the connection to the database and the database interactions (part 10 of 10).

Line 43 of the CloudscapeDataAccess constructor (lines 40–127) invokes utility method connect (defined at lines 134–152) to perform the connection to the database.
Any exceptions that occur in method `connect` are thrown back to class `AddressBook`, so the application can determine an appropriate course of action to take for a failed connection. If the connection is successful, lines 46–126 invoke `Connection` method `prepareStatement` to create each of the SQL statements that manipulate database `addressbook`. These `PreparedStatement`s perform standard `SELECT`, `INSERT`, `UPDATE` and `DELETE` operations, as discussed in Section 8.4. The question marks (?) in each `PreparedStatement` represent the parameters that must be set before the program executes each statement.

`PreparedStatement sqlFind` (lines 46–55) selects all the data for a person with a specific `lastName` from the four tables in database `addressbook`. Note that the `WHERE` clause uses `AND`ed conditions to ensure that the query retrieves the appropriate data from each table. These conditions compare the `personID` fields in each table. The only records this query returns are those with the specified last name where the `personID` field in table `names` matches the `personID` field in tables `addresses`, `phoneNumbers` and `emailAddresses`. [Note: As implemented, this application assumes that all last names are unique.]

`PreparedStatement sqlPersonID` (lines 59–61) is a Cloudscape-specific operation to determine the last autoincrement value for the `names` table’s `personID` field. `ConnectionInfo.lastAutoincrementValue` is a static Java method built into Cloudscape. The method receives three SQL string arguments that represent the name of the database schema (`'APP'`) containing the table, the name of the table containing the autoincrement field (`'NAMES'`) and the name of the autoincrement field (`'PERSONID'`). Cloudscape requires each of these names to be in all uppercase letters. In a database, it is possible to group tables into sets of tables with a specific schema name. SQL statements can qualify table names with schema names to interact with tables from different schemas in the same SQL statement. Cloudscape places database tables in schema `APP` by default.

The remaining `PreparedStatement`s are straightforward. For more detail on the SQL used in those statements, refer back to Section 8.4.

`CloudscapeDataAccess` method `connect` (lines 134–152) establishes the connection to database `addressbook` with the techniques shown earlier in this chapter. Any exceptions that occur while attempting to load the database driver and connect to the database are thrown from this method back to the called method (the constructor). The database connection in this example differs from those in prior examples in that line 151 disables automatic commitment of transactions. Thus, the program must indicate when a transaction should be committed to the database or rolled back to maintain the database’s state before the transaction. This enables `CloudscapeDataAccess` to execute a series of SQL statements and commit the results only if all the statements in the series are successful.

`CloudscapeDataAccess` method `findPerson` (lines 156–196) receives a `String` containing the last name of the person to locate in the database and uses that last name to set the parameter in `PreparedStatement sqlFind` (line 160). Line 161 executes `sqlFind`. If a record is found, lines 168–186 set the properties of the `AddressBookEntry`, and line 189 returns the `AddressBookEntry`. If no records are found for the specified last name, the method returns `null`.

`CloudscapeDataAccess` method `savePerson` (lines 200–274) receives an `AddressBookEntry` containing the complete information for a person to update in the
database and uses that information to set the parameters of `PreparedStatement sqlUpdateName` (lines 208–210), `sqlUpdateAddress` (lines 220–225), `sqlUpdatePhone` (lines 235–236) and `sqlUpdateEmail` (lines 246–247). Note that parameter values are set by invoking `PreparedStatement set` methods for the appropriate data type. In this example, the ID parameters are all integers and the remaining data are all strings, so the program uses methods `setInt` and `setString` to specify parameters. After setting the parameters for a particular `PreparedStatement`, the method calls that statement’s `executeUpdate` method (lines 211, 226, 237 and 248), which returns an integer indicating the number of rows modified by the update. The execution of each `PreparedStatement` is followed by an `if` structure that tests the return value of `executeUpdate`. If `executeUpdate` returns 0, the `PreparedStatement` did not update any records. Therefore, `savePerson` invokes `Connection method rollback` to restore the database to its state before the `PreparedStatement` executed and returns `false` to indicate to the `AddressBook` application that the update failed. If `savePerson` reaches line 256, it commits the transaction in the database and returns `true` to indicate that the update was successful.

`CloudscapeDataAccess` method `newPerson` (lines 278–367) is similar to method `savePerson`. Method `newPerson` receives an `AddressBookEntry` containing the complete information for a person to insert in the database and uses that information to set the parameters of `PreparedStatement sqlInsertName` (lines 286–287), `sqlInsertAddress` (lines 303–313), `sqlInsertPhone` (lines 323–325) and `sqlInsertEmail` (lines 335–337). The primary difference between `newPerson` and `savePerson` is that the entry does not exist in the database yet. To insert rows in tables `addresses`, `phoneNumbers` and `emailAddresses`, the `personID` foreign-key field for each new record must correspond to the `personID` primary-key field in the `names` table. The new `personID` in table `names` is not known until the program inserts the new record in the table. So, after inserting a new record into table `names`, line 297 executes `PreparedStatement sqlPersonID` to obtain the `personID` number for the last new person added to table `names`. Line 300 places this value in the local variable `personID`. Then, the program inserts records in tables `addresses`, `phoneNumbers` and `emailAddresses`, using the new `personID` as the value of the foreign-key field in each table. As in method `savePerson`, if no records are inserted after a given `PreparedStatement` executes, method `newPerson` rolls back the transaction and returns `false` to indicate that the insertion failed. Otherwise, method `newPerson` commits the transaction and returns `true` to indicate that the insertion succeeded.

`CloudscapeDataAccess` method `deletePerson` (lines 371–435) receives an `AddressBookEntry` containing the `personID` of the person to remove from the database and uses that ID as the parameter value for the `PreparedStatement sqlDeleteName`, `sqlDeleteAddress`, `sqlDeletePhone` and `sqlDeleteEmail`. When each `PreparedStatement` executes, it deletes all records with the specified `personID` in the appropriate table. If any part of the delete fails, method `deletePerson` rolls back the transaction and returns `false` to indicate that the deletion failed. Otherwise, method `deletePerson` commits the transaction and returns `true` to indicate that the deletion succeeded. In the future, if this program supports multiple addresses, phone numbers and e-mail addresses for each person, this `deletePerson` method will delete all the information for a particular entry properly.
CloudscapeDataAccess methods close (lines 438–463) and finalize (lines 467–470) close the PreparedStatements and database connection. Method finalize is provided in case an object of class CloudscapeDataAccess gets garbage collected and the client forgot to call close explicitly.

Class AddressBookEntryFrame (Fig. 8.37) is a subclass of JInternalFrame that enables address-book application users to view or edit the details of an AddressBookEntry. The AddressBook application class (Fig. 8.38) creates a new AddressBookEntryFrame to display the results of a search for an entry and to enable the user to input information for a new entry. AddressBookEntryFrame maintains a reference to the currently displayed AddressBookEntry and provides set and get methods to specify an AddressBookEntry to display and to return the currently displayed AddressBookEntry, respectively. The class also has several private utility methods for setting up the GUI and accessing the individual JTextFields in the GUI. Objects of class AddressBookEntryFrame are managed by class AddressBook, which contains a JDesktopPane.

```
// Fig. 8.37: AddressBookEntryFrame.java
// A subclass of JInternalFrame customized to display and
// an AddressBookEntry or set an AddressBookEntry's properties
// based on the current data in the UI.
package com.deitel.advjhtp1.jdbc.addressbook;

// Java core packages
import java.util.*;
import java.awt.*;

// Java extension packages
import javax.swing.*;

public class AddressBookEntryFrame extends JInternalFrame {

    // HashMap to store JTextField references for quick access
    private HashMap fields;

    // current AddressBookEntry set by AddressBook application
    private AddressBookEntry person;

    // panels to organize GUI
    private JPanel leftPanel, rightPanel;

    // static integers used to determine new window positions
    // for cascading windows
    private static int xOffset = 0, yOffset = 0;

    // static Strings that represent name of each text field.
    // These are placed on JLabels and used as keys in
    // HashMap fields.
    private static final String FIRST_NAME = "First Name",
```
LAST_NAME = "Last Name", ADDRESS1 = "Address 1",
ADDRESS2 = "Address 2", CITY = "City", STATE = "State",
ZIPCODE = "Zipcode", PHONE = "Phone", EMAIL = "Email";

// construct GUI
public AddressBookEntryFrame()
{
    super( "Address Book Entry", true, true );
    fields = new HashMap();
    leftPanel = new JPanel();
    leftPanel.setLayout( new GridLayout( 9, 1, 0, 5 ) );
    rightPanel = new JPanel();
    rightPanel.setLayout( new GridLayout( 9, 1, 0, 5 ) );
    createRow( FIRST_NAME );
    createRow( LAST_NAME );
    createRow( ADDRESS1 );
    createRow( ADDRESS2 );
    createRow( CITY );
    createRow( STATE );
    createRow( ZIPCODE );
    createRow( PHONE );
    createRow( EMAIL );

    Container container = getContentPane();
    container.add( leftPanel, BorderLayout.WEST );
    container.add( rightPanel, BorderLayout.CENTER );

    setBounds( xOffset, yOffset, 300, 300 );
    xOffset = ( xOffset + 30 ) % 300;
    yOffset = ( yOffset + 30 ) % 300;
}

// set AddressBookEntry then use its properties to
// place data in each JTextField
public void setAddressBookEntry( AddressBookEntry entry )
{
    person = entry;

    setField( FIRST_NAME, person.getFirstName() );
    setField( LAST_NAME, person.getLastName() );
    setField( ADDRESS1, person.getAddress1() );
    setField( ADDRESS2, person.getAddress2() );
    setField( CITY, person.getCity() );
    setField( STATE, person.getState() );
    setField( ZIPCODE, person.getZipcode() );
    setField( PHONE, person.getPhoneNumber() );
    setField( EMAIL, person.getEmailAddress() );
}
// store AddressBookEntry data from GUI and return AddressBookEntry
public AddressBookEntry getAddressBookEntry()
{
    person.setFirstName( getField( FIRST_NAME ) );
    person.setLastName( getField( LAST_NAME ) );
    person.setAddress1( getField( ADDRESS1 ) );
    person.setAddress2( getField( ADDRESS2 ) );
    person.setCity( getField( CITY ) );
    person.setState( getField( STATE ) );
    person.setZipcode( getField( ZIPCODE ) );
    person.setPhoneNumber( getField( PHONE ) );
    person.setEmailAddress( getField( EMAIL ) );

    return person;
}

// set text in JTextField by specifying field's name and value
private void setField( String fieldName, String value )
{
    JTextField field =
    (JTextField) fields.get( fieldName );

    field.setText( value );
}

// get text in JTextField by specifying field's name
private String getField( String fieldName )
{
    JTextField field =
    (JTextField) fields.get( fieldName );

    return field.getText();
}

// utility method used by constructor to create one row in GUI containing JLabel and JTextField
private void createRow( String name )
{
    JLabel label = new JLabel( name, SwingConstants.RIGHT );
    label.setBorder(
        BorderFactory.createEmptyBorder( 5, 5, 5, 5 ) );
    leftPanel.add( label );

    JTextField field = new JTextField( 30 );
    rightPanel.add( field );

    fields.put( name, field );
}

} // end class AddressBookEntryFrame

Fig. 8.37  AddressBookEntryFrame for viewing and editing an AddressBookEntry (part 3 of 3).
Class **AddressBook** (Fig. 8.38) is the main application class for the address-book application. **AddressBook** uses several of the GUI techniques presented in Chapter 2, including tool bars, menus, actions and multiple-document interfaces. The discussion of class **AddressBook** concentrates on the functionality, rather than on the GUI details. Screen captures demonstrating the program’s execution appear in Fig. 8.39.

```java
// Fig. 8.38: AddressBook.java
// An address book database example that allows information to
// be inserted, updated and deleted. The example uses
// transactions to ensure that the operations complete
// successfully.
package com.deitel.advjhtp1.jdbc.addressbook;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.sql.*;

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

public class AddressBook extends JFrame {
    // reference for manipulating multiple document interface
    private JDesktopPane desktop;
    // reference to database access object
    private AddressBookDataAccess database;
    // references to Actions
    Action newAction, saveAction, deleteAction,
    searchAction, exitAction;
    // set up database connection and GUI
    public AddressBook()
    {
        super( "Address Book" );
        // create database connection
        try {
            database = new CloudscapeDataAccess();
        }
        // detect problems with database connection
        catch ( Exception exception ) {
            exception.printStackTrace();
            System.exit( 1 );
        }
    }
}
```

Fig. 8.38 **AddressBook** application class that enables the user to interact with the **addressbook** database (part 1 of 8).
45  // database connection successful, create GUI
46  JToolBar toolBar = new JToolBar();
47  JMenu fileMenu = new JMenu( "File" );
48  fileMenu.setMnemonic( 'F' );
49
50  // Set up actions for common operations. Private inner
51  // classes encapsulate the processing of each action.
52  newAction = new NewAction();
53  saveAction = new SaveAction();
54  saveAction.setEnabled( false );    // disabled by default
55  deleteAction = new DeleteAction();
56  deleteAction.setEnabled( false );  // disabled by default
57  searchAction = new SearchAction();
58  exitAction = new ExitAction();
59
60  // add actions to tool bar
61  toolBar.add( newAction );
62  toolBar.add( saveAction );
63  toolBar.add( deleteAction );
64  toolBar.add( new JToolBar.Separator() );
65  toolBar.add( searchAction );
66
67  // add actions to File menu
68  fileMenu.add( newAction );
69  fileMenu.add( saveAction );
70  fileMenu.add( deleteAction );
71  fileMenu.addSeparator();
72  fileMenu.add( searchAction );
73  fileMenu.addSeparator();
74  fileMenu.add( exitAction );
75
76  // set up menu bar
77  JMenuBar menuBar = new JMenuBar();
78  menuBar.add( fileMenu );
79  setJMenuBar( menuBar );
80
81  // set up desktop
82  desktop = new JDesktopPane();
83
84  // get the content pane to set up GUI
85  Container c = getContentPane();
86  c.add( toolBar, BorderLayout.NORTH );
87  c.add( desktop, BorderLayout.CENTER );
88
89  // register for windowClosing event in case user
90  // does not select Exit from File menu to terminate
91  // application
92  addWindowListener(
93      new WindowAdapter() {
94          public void windowClosing( WindowEvent event )
95          {
96              shutDown();
97          }
98      } );

Fig. 8.38 AddressBook application class that enables the user to interact with the addressbook database (part 2 of 8).
private void shutDown() {
    database.close();  // close database connection
    System.exit( 0 );  // terminate program
}

private AddressBookEntryFrame createAddressBookEntryFrame() {
    AddressBookEntryFrame frame = new AddressBookEntryFrame();
    setDefaultCloseOperation( DISPOSE_ON_CLOSE );
    frame.addInternalFrameListener( new InternalFrameAdapter() {

        // internal frame becomes active frame on desktop
        public void internalFrameActivated( InternalFrameEvent event ) {
            saveAction.setEnabled( true );
            deleteAction.setEnabled( true );
        }

        // internal frame becomes inactive frame on desktop
        public void internalFrameDeactivated( InternalFrameEvent event ) {
            saveAction.setEnabled( false );
            deleteAction.setEnabled( false );
        }
    } );  // end call to addInternalFrameListener

    return frame;
}  // end method createAddressBookEntryFrame

Fig. 8.38 AddressBook application class that enables the user to interact with the addressbook database (part 3 of 8).
// method to launch program execution
public static void main( String args[] )
{
    new AddressBook();
}

// Private inner class defines action that enables
// user to input new entry. User must "Save" entry
// after inputting data.
private class NewAction extends AbstractAction {

    // set up action's name, icon, descriptions and mnemonic
    public NewAction()
    {
        putValue( NAME, "New" );
        putValue( SMALL_ICON, new ImageIcon(
            getClass().getResource( "images/New24.png" ) ) );
        putValue( SHORT_DESCRIPTION, "New" );
        putValue( LONG_DESCRIPTION,
            "Add a new address book entry" );
        putValue( MNEMONIC_KEY, new Integer( 'N' ) );
    }

    // display window in which user can input entry
    public void actionPerformed( ActionEvent e )
    {
        // create new internal window
        AddressBookEntryFrame entryFrame = createAddressBookEntryFrame();

        // set new AddressBookEntry in window
        entryFrame.setAddressBookEntry(
            new AddressBookEntry() );

        // display window
        desktop.add( entryFrame );
        entryFrame.setVisible( true );
    }
}

// inner class defines an action that can save new or
// updated entry
private class SaveAction extends AbstractAction {

    // set up action's name, icon, descriptions and mnemonic
    public SaveAction()
    {
        putValue( NAME, "Save" );
        putValue( SMALL_ICON, new ImageIcon(
            getClass().getResource( "images/Save24.png" ) ) );
        putValue( SHORT_DESCRIPTION, "Save" );
    }
}

Fig. 8.38 AddressBook application class that enables the user to interact with the addressbook database (part 4 of 8).
Chapter 8 Java Database Connectivity (JDBC) 507

200    putValue( LONG_DESCRIPTION,  
201          "Save an address book entry" );  
202    putValue( MNEMONIC_KEY, new Integer( 'S' ) );  
203
204     // save new entry or update existing entry
205    public void actionPerformed( ActionEvent e )  
206    {
207        // get currently active window
208        AddressBookEntryFrame currentFrame =  
209            ( AddressBookEntryFrame ) desktop.getSelectedFrame();  
210
211        // obtain AddressBookEntry from window
212        AddressBookEntry person =  
213            currentFrame.getAddressBookEntry();  
214
215        // insert person in address book
216        try {  
217            try {  
218                // Get personID. If 0, this is a new entry;
219                // otherwise an update must be performed.
220                int personID = person.getPersonID();  
221
222                // determine string for message dialogs
223                String operation =  
224                    ( personID == 0 ) ? "Insertion" : "Update";  
225
226                // insert or update entry
227                if ( personID == 0 )  
228                    database.newPerson( person );  
229                else  
230                    database.savePerson( person );  
231
232                // display success or failure message
233                JOptionPane.showMessageDialog( desktop,  
234                    operation + " successful" );  
235            }  // end try
236        }  // end method actionPerformed
237
238        // detect database errors
239        catch ( DataAccessException exception ) {  
240            JOptionPane.showMessageDialog( desktop, exception,  
241                "DataAccessException",  
242                JOptionPane.ERROR_MESSAGE );  
243            exception.printStackTrace();  
244        }
245
246        // close current window and dispose of resources
247        currentFrame.dispose();  
248
249    }  // end inner class SaveAction

Fig. 8.38 AddressBook application class that enables the user to interact with the addressbook database (part 5 of 8).
private class DeleteAction extends AbstractAction {

    // set up action’s name, icon, descriptions and mnemonic
    public DeleteAction() {
        putValue(NAME, "Delete");
        putValue(SMALL_ICON, new ImageIcon(getClass().getResource("images/Delete24.png")));
        putValue(SHORT_DESCRIPTION, "Delete");
        putValue(LONG_DESCRIPTION, "Delete an address book entry");
        putValue(MNEMONIC_KEY, new Integer('D'));
    }

    // delete entry
    public void actionPerformed(ActionEvent e) {
        // get currently active window
        AddressBookEntryFrame currentFrame = (AddressBookEntryFrame) desktop.getSelectedFrame();

        // get AddressBookEntry from window
        AddressBookEntry person = currentFrame.getAddressBookEntry();

        // If personID is 0, this is new entry that has not
        // been inserted. Therefore, delete is not necessary.
        // Display message and return.
        if (person.getPersonID() == 0) {
            JOptionPane.showMessageDialog(desktop,
                "New entries must be saved before they can be " +
                "deleted. \nTo cancel a new entry, simply " +
                "close the window containing the entry");
            return;
        }

        // delete person
        try {
            database.deletePerson(person);

            // display message indicating success
            JOptionPane.showMessageDialog(desktop,
                "Deletion successful");
        }

        // detect problems deleting person
        catch (DataAccessException exception) {
            JOptionPane.showMessageDialog(desktop, exception,
                "Deletion failed", JOptionPane.ERROR_MESSAGE);
        }
    }

Fig. 8.38 AddressBook application class that enables the user to interact with the
addressbook database (part 6 of 8).
exception.printStackTrace();

// close current window and dispose of resources
currentFrame.dispose();

} // end method actionPerformed

} // end inner class DeleteAction

// inner class defines action that locates entry
private class SearchAction extends AbstractAction {
  
  // set up action's name, icon, descriptions and mnemonic
  public SearchAction()
  {
    putValue( NAME, "Search" );
    putValue( SMALL_ICON, new ImageIcon( 
      getClass().getResource( "images/Find24.png" ) ) );
    putValue( SHORT_DESCRIPTION, "Search" );
    putValue( LONG_DESCRIPTION, "Search for an address book entry" );
    putValue( MNEMONIC_KEY, new Integer( 'r' ) );
  }

  // locate existing entry
  public void actionPerformed( ActionEvent e )
  {
    String lastName =
      JOptionPane.showInputDialog( desktop,
      "Enter last name" );

    // if last name was input, search for it; otherwise,
    // do nothing
    if ( lastName != null ) {
      AddressBookEntry person = database.findPerson( lastName );

      if ( person != null ) {
        AddressBookEntryFrame entryFrame =
          createAddressBookEntryFrame();

        entryFrame.setAddressBookEntry( person );

        desktop.add( entryFrame );
      }
    }
  }

Fig. 8.38 AddressBook application class that enables the user to interact with the addressbook database (part 7 of 8).
Fig. 8.38 AddressBook application class that enables the user to interact with the addressbook database (part 8 of 8).

Class AddressBook’s constructor (lines 30–110) creates a Cloudscape-DataAccess object to interact with the database (line 36), builds the GUI (lines 46–87), registers an event handler for the window-closing event (lines 92–99) and displays the application window (lines 102–109). As part of building the tool bar and menu for the application, lines 52–58 of the constructor create instances of five private inner classes that implement the actions for the GUI—NewAction (lines 157–187), SaveAction (lines 191–251), DeleteAction (lines 254–311), SearchAction (lines 314–366) and ExitAction (lines 370–387). Note that the program disables the SaveAction and DeleteAction by default. These are enabled only if there is an active internal frame on the desktop.

Each action (except ExitAction) uses a standard icon from the Sun Microsystems Java Look and Feel Graphics Repository, located at developer.java.sun.com/developer/techDocs/hi/repository. The first screen capture of Fig. 8.39 describes each of the icons in the GUI.
Fig. 8.39 Screen captures of the **AddressBook** application (part 1 of 3).
Fig. 8.39  Screen captures of the `AddressBook` application (part 2 of 3).
Fig. 8.39 Screen captures of the AddressBook application (part 3 of 3).
NewAction (lines 157–187) does not perform any database manipulations. It simply displays an AddressBookEntryFrame in which the user inputs the information for a new address book entry. To perform the actual insert into the database, the user must click the Save button or select Save from the File menu, which invokes the SaveAction. Class NewAction's actionPerformed method (lines 172–185) creates a new AddressBookEntryFrame (lines 175–176), sets a new AddressBookEntry for the frame (lines 179–180), attaches the frame to the JDesktopPane (line 183) and displays the frame (line 184).

SaveAction (lines 191–251) determines whether to save a new entry or update an existing entry based on the personID for the AddressBookEntry in the currently active internal frame. Method actionPerformed (lines 206–249) obtains a reference to the active internal frame (lines 209–210) and gets the AddressBookEntry currently displayed (lines 213–214). Line 221 gets the personID from the AddressBookEntry. If the personID is 0, the AddressBookEntry represents a new address book entry, and line 229 invokes the CloudscapeDataAccess object's newPerson method to insert a new record in the database. If the personID is not 0, the AddressBookEntry represents an existing address book entry to update, and line 231 invokes the CloudscapeDataAccess object's savePerson method to update the record in the database. Methods newPerson and savePerson each receive an AddressBookEntry as an argument. Line 247 disposes of the active internal frame after the save operation completes.

DeleteAction (lines 254–311) uses the AddressBookEntry in the currently active internal frame to remove an entry from the database. Method actionPerformed (lines 269–309) obtains a reference to the active internal frame (lines 272–273) and gets the currently displayed AddressBookEntry (lines 276–277). If the personID in the AddressBookEntry is 0, the entry has not been stored in the database, so actionPerformed displays a message to the user and terminates (lines 282–288). Line 292 invokes the CloudscapeDataAccess object’s deletePerson method, passing the AddressBookEntry to delete as an argument. Line 307 disposes of the active internal frame after the delete operation completes.

SearchAction (lines 314–366) searches for an address book entry based on the last name of the person input by the user. Method actionPerformed (lines 329–364) obtains the last name for which to search (lines 331–333). If the last name is not null (i.e., the user did not click the Cancel button in the input dialog), lines 341–342 create a new AddressBookEntry reference and invokes database’s findPerson method to locate the person in the database. If the person exists, findPerson returns the AddressBookEntry containing the information for that person. Then, actionPerformed creates a new AddressBookEntryFrame (lines 347–348), sets the AddressBookEntry for the frame (line 351), attaches the frame to the JDesktopPane (line 354) and displays the frame (line 355). Otherwise, actionPerformed displays a message dialog indicating that the record was not found.

In Fig. 8.39, the first screen capture shows the address-book application after the user clicks the New button to create a new entry. The second screen capture shows the results after the user inputs the information for the new entry and clicks the Save button to insert the data in the database. The third and fourth screen captures demonstrate searching for an entry in the database. The fifth screen capture demonstrates updating the person’s city information. The sixth screen capture demonstrates deleting the record for the currently dis-
played entry. [\textit{Note: The screen captures do not show that after completing a Save or Delete operation, the internal frame that displays the entry is removed from the screen.}]

\section*{8.8 Stored Procedures}

Many database management systems can store individual SQL statements or sets of SQL statements in a database, so that programs accessing that database can invoke them. Such SQL statements are called \textit{stored procedures}. JDBC enables programs to invoke stored procedures using objects that implement interface \texttt{CallableStatement}. Like \texttt{PreparedStatements}, \texttt{CallableStatements} can receive arguments specified with the methods inherited from interface \texttt{PreparedStatement}. In addition, \texttt{CallableStatements} can specify \textit{output parameters} in which a stored procedure can place return values. Interface \texttt{CallableStatement} includes methods to specify which parameters in a stored procedure are output parameters. The interface also includes methods to obtain the values of output parameters returned from a stored procedure.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8_40.png}
\caption{\texttt{Statement} and \texttt{PreparedStatement} methods for batch updates (part 1 of 2).}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Method} & \textbf{Description} \\
\hline
\texttt{public void addBatch( String sql )} & Method of interface \texttt{Statement} that receives a \texttt{String} argument specifying an SQL statement to add to the \texttt{Statement}’s batch for future execution. This method should not be used with \texttt{PreparedStatements} and \texttt{CallableStatements}. \\
\hline
\end{tabular}
\end{table}

\textbf{Portability Tip 8.8}

\textit{Although the syntax for creating stored procedures differs across database management systems, interface \texttt{CallableStatement} provides a uniform interface for specifying input and output parameters for stored procedures and for invoking stored procedures.}

\textbf{Portability Tip 8.9}

\textit{According to the Java API documentation for interface \texttt{CallableStatement}, for maximum portability between database systems, programs should process the update counts or \texttt{ResultSets} returned from a \texttt{CallableStatement} before obtaining the values of any output parameters.}

\section*{8.9 Batch Processing}

A series of database updates (e.g., inserts, updates, deletes) can be performed in a \textit{batch update} to the database. JDBC \texttt{Statements}, \texttt{PreparedStatement}s and \texttt{CallableStatement}s provide an \texttt{addBatch} method that enables the program to add SQL statements to a batch for future execution. Each \texttt{Statement}, \texttt{PreparedStatement} or \texttt{CallableStatement} object maintains its own list of SQL statements to perform in a batch update. Figure 8.40 describes the batch-processing methods of interfaces \texttt{Statement} and \texttt{PreparedStatement}. (\texttt{CallableStatement}s inherit the methods of interface \texttt{PreparedStatement}.)
Each method in Fig. 8.40 throws a `BatchUpdateException` (a subclass of `SQLException`) if database errors occur while executing any of the SQL statements or if the particular database management system does not support batch update processing. Method `executeBatch` also throws `BatchUpdateExceptions` if the batch update contains any SQL statements that return `ResultSets`.

**Common Programming Error 8.14**

Batch updates are for use only with SQL statements that do not return `ResultSets`. Executing an SQL statement that returns a `ResultSet` as part of a batch update causes a `BatchUpdateException`.

After adding statements to a batch update, a program invokes `Statement` method `executeBatch` to execute the SQL statements in the batch. This method performs each SQL statement and returns an array of `int` values indicating the status of each SQL statement in the batch. The order of the values in the array corresponds to the order in which the SQL statements are added to the batch.

---

**Method** | **Description**
--- | ---
`public void addBatch()` | Method of interface `PreparedStatement` that adds the statement to a batch for future execution. This method should be called after setting the parameters for the `PreparedStatement`. This version of `addBatch` also can be used with `CallableStatements`.

`public void clearBatch()` | Method of interface `Statement` that clears the statement’s batch.

`public int[] executeBatch()` | Method of interface `Statement` that executes the statement’s batch. The method returns an array of `int` values indicating the status of each SQL statement in the batch. The order of the values in the array corresponds to the order in which the SQL statements are added to the batch.

---

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a value greater than or equal to 0</td>
<td>Indicates successful execution of the SQL statements in the batch update. The value specifies the actual number of rows updated in the database.</td>
</tr>
<tr>
<td>-2</td>
<td>Indicates successful execution of the SQL statements in the batch update and that the affected number of rows is unknown.</td>
</tr>
</tbody>
</table>

---

**Fig. 8.40** `Statement` and `PreparedStatement` methods for batch updates (part 2 of 2).

**Fig. 8.41** Return values of method `executeBatch` (part 1 of 2).
Normally, programs disable autocommit mode for a Connection before executing a batch update. Otherwise, each SQL statement in the batch update is committed individually, which prevents programs from deciding whether groups of SQL statements should be committed or rolled back, based on logic in the program.

8.10 Processing Multiple ResultSets or Update Counts

Some Statements, PreparedStatements and CallableStatements return multiple ResultSets or update counts. In such cases, programs should use Statement method execute to execute the SQL statements. After executing SQL statements, method execute returns a boolean indicating whether the first result is a ResultSet (true) or an update count (false). Based on execute’s return value, the program invokes method getResultSet or method getUpdateCount to obtain the first result. The program can obtain subsequent results by calling method getNextResults. Figure 8.42 summarizes the methods for processing multiple results. [Note: Each of the methods in Fig. 8.42 is defined in interface Statement and inherited into interfaces PreparedStatement and CallableStatement.]

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean execute()</td>
<td>Programs use this method to execute SQL statements that can return multiple ResultSet or update counts. This method returns a boolean indicating whether the first result is a ResultSet (true) or an update count (false). Based on the value returned, the program can call getResultSet or getUpdateCount to obtain the first result.</td>
</tr>
</tbody>
</table>
Software Engineering Observation 8.13

A program has completed processing the results returned by method `execute` when method `getMoreResults` returns `false` and method `getUpdateCount` returns `-1`.

8.11 Updatable ResultSets

Some JDBC drivers support *updatable* `ResultSet`s. Such `ResultSet`s enable a program to insert, update and delete records using methods of interface `ResultSet`. If the JDBC driver supports updatable `ResultSet`s, the program can invoke `Connection` method `createStatement`, `prepareStatement` or `prepareCall` and specify the constant `ResultSet.CONCUR_UPDATABLE` as the second argument (the first argument specifies the type of scrolling supported by the `ResultSet`).

Software Engineering Observation 8.14

Normally, a query that produces an updatable `ResultSet` must select a table’s primary key, so that updates can determine the proper records to manipulate in the database. Otherwise, the query returns a read-only `ResultSet`.

Interface `ResultSet` provides update methods that enable the program to specify new values for particular columns in the current `ResultSet` row. In addition, interface `ResultSet` provides methods `deleteRow`, `insertRow` and `updateRow` to manipulate the `ResultSet` and the underlying database. Method `deleteRow` deletes the current `ResultSet` row from the `ResultSet` and the database. Method `updateRow` updates the current row in the `ResultSet` and the database. Method `insertRow` inserts a new row in the `ResultSet` and the database. Every updatable `ResultSet` maintains

---

### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public boolean getMoreResults()</code></td>
<td>After obtaining the first result returned from method <code>execute</code>, a program</td>
</tr>
<tr>
<td></td>
<td>invokes this method to move to the next result. This method returns a <code>bool-</code></td>
</tr>
<tr>
<td></td>
<td>ean indicating whether the next result is a <code>ResultSet</code> (<code>true</code>) or an</td>
</tr>
<tr>
<td></td>
<td>update count (<code>false</code>). Based on the value returned, the program can call</td>
</tr>
<tr>
<td></td>
<td><code>getResultSet</code> or <code>getUpdateCount</code> to obtain the next result.</td>
</tr>
<tr>
<td><code>public ResultSet getResultSet()</code></td>
<td>Obtains a <code>ResultSet</code> from the results returned by method <code>execute</code>. This</td>
</tr>
<tr>
<td></td>
<td>method returns <code>null</code> if the result is not a <code>ResultSet</code> or if there are no</td>
</tr>
<tr>
<td></td>
<td>more results.</td>
</tr>
<tr>
<td><code>public int getUpdateCount()</code></td>
<td>Obtains an update count from the results returned by method <code>execute</code>. This</td>
</tr>
<tr>
<td></td>
<td>method returns <code>-1</code> if the result is not an update count or if there are no</td>
</tr>
<tr>
<td></td>
<td>more results.</td>
</tr>
</tbody>
</table>

---

Fig. 8.42 Statement methods that enable processing of multiple results returned by method `execute` (part 2 of 2).
an insert row where the program can build a new record before inserting it in the ResultSet and the database. Before invoking ResultSet’s update methods to build the new record, the program must invoke ResultSet method moveToInsertRow. The ResultSet keeps track of the cursor’s location before that operation. The program can return to the cursor location to continue processing the ResultSet by invoking ResultSet method moveToCurrentRow.

### 8.12 JDBC 2.0 Optional Package javax.sql

In addition to the classes and interfaces of package java.sql, many database vendors now support the JDBC 2.0 optional package javax.sql. Typically, this package is included with the implementation of the Java 2 Enterprise Edition. Some of the key interfaces in package javax.sql include DataSource, ConnectionPoolDataSource, PooledConnection and RowSet. Each of these interfaces is explained briefly in the next several subsections.

#### 8.12.1 DataSource

A DataSource is new way for programs to obtain database connections. Enterprise Java applications often access information and resources (such as databases) that are external to those applications. In some cases, resources are distributed across a network. Enterprise application components must be able to locate the resources they use. An Enterprise Java application container must provide a naming service that implements the Java Naming and Directory Interface (JNDI) and enables the components executing in that container to perform name lookups to locate resources. Typically, DataSources are registered with a JNDI service that enables a program to locate the DataSource. Chapter 11 demonstrates our first Enterprise Java application that uses a JNDI service to look up a DataSource and connect to a database.

#### 8.12.2 Connection Pooling

The process of connecting to a database requires substantial overhead in both time and resources. In a program that performs many separate database connections (such as a server in a Web-based shopping-cart application), such overhead can become a burden on the program. Applications can establish connection pools that maintain many database connections to eliminate the overhead of connecting to the database while many clients are waiting for responses in distributed applications. These connection objects can be shared between the application clients.

Databases that provide full support for the JDBC optional package include implementations of interfaces ConnectionPoolDataSource and PooledConnection. Like DataSources ConnectionPoolDataSources typically are registered with a JNDI service, so that a program can locate them dynamically. After obtaining a reference to a ConnectionPoolDataSource, a program can invoke its getPooledConnection method to obtain a PooledConnection object that represents the connection to the database. PooledConnection method getConnection returns the underlying Connection object that the program uses to create Statements, PreparedStatements and CallableStatements for executing SQL statements.
8.12.3 RowSets

The JDBC optional package introduces a new interface, `RowSet`, for manipulating tabular data sources such as `ResultSet`s. `RowSet`s are not implemented as part of the database driver. Instead, they are implemented as JavaBeans that encapsulate a tabular data source.

Interface `RowSet` extends interface `ResultSet`. Thus, a `RowSet` object has all the functionality of a `ResultSet`, including the ability to scroll through the records, insert new records, update existing records and delete existing records. What makes a `RowSet` interesting is that all of these features are supported regardless of whether the `ResultSet` implementation provided by a particular database driver supports these features. For example, a program can create a `RowSet` based on a `ResultSet`, disconnect from the database and allow the program to update the data in the `RowSet`. Then the `RowSet` can connect to the database and update it, based on the changes made to the data in the `RowSet`. All of this can be accomplished even if the database driver does not support updatable `ResultSet`s. `RowSet`s that disconnect from the database and then reconnect to perform updates are called `disconnected RowSet`s.

Unlike `ResultSet`s, `RowSet`s implementations can be serializable, so they can be saved locally or transmitted across a network. `RowSet`s also support JavaBean events (with interface `RowSetListener` and class `RowSetEvent`) that enable an application using a `RowSet` to be notified when the `RowSet` cursor moves, a record is inserted, a record is updated, a record is deleted or the entire set of data in the `RowSet` changes. `RowSet`s also allow a program to set parameters to the `RowSet`’s `command string`—normally, the SQL statement that obtains the data.

If you are interested in experimenting with `RowSet`s, Sun Microsystems has three early access `RowSet` implementations available at

```
developer.java.sun.com/developer/earlyAccess/crs
```

These implementations are `CachedRowSet`, `WebRowSet` and `JDBCRowSet`.

Class `CachedRowSet` defines a disconnected `RowSet` that can be serialized. `CachedRowSet` provides full support for scrolling through data and updating data—both particularly useful in applications that require these capabilities even if the database driver’s `ResultSet` implementation does not support scrolling through and updating data. The fact that `CachedRowSet`s are serializable enables them to be sent across a network in a distributed network application.

Class `WebRowSet` is a subclass of `CachedRowSet` that enables `RowSet` data to be output as an XML document. Class `JDBCRowSet` defines a connected `RowSet` that encapsulates a `ResultSet` to make the `ResultSet` appear like a JavaBean to the program.

For a tutorial on using the Sun Microsystems, Inc., `RowSet` implementations, visit

```
developer.java.sun.com/developer/Books/JDBCTutorial/chapter5.html
```

Also, check your database management system’s documentation to determine whether your database provides any `RowSet` implementations.

8.13 Internet and World Wide Web Resources

```
java.sun.com/products/jdbc
```

Sun Microsystems, Inc.’s JDBC home page.
SUMMARY

- A database is an integrated collection of data. A database management system (DBMS) provides mechanisms for storing and organizing data.
- Today’s most popular database systems are relational databases.
- A language called Structured Query Language (SQL) is used almost universally with relational database systems to perform queries and manipulate data.
- A programming language connects to, and interacts with, relational databases via an interface—software that facilitates communications between a database management system and a program.
- Java programmers communicate with databases and manipulate their data using the Java Database Connectivity (JDBC) API. A JDBC driver implements the interface to a particular database.
- A relational database is composed of tables. A row of a table is called a record (or row).
- A primary key is a field that contains unique data that cannot be duplicated in other records.
- Each column of the table represents a different field (or column or attribute).
- The primary key can be composed of more than one column (or field) in the database.
- SQL provides a complete set of commands that enable programmers to define complex queries that select data from a table. The results of a query are commonly called result sets (or record sets).
- Every record must have a value in the primary-key field, and that value must be unique. This is known as the Rule of Entity Integrity.
• A one-to-many relationship between tables indicates that a record in one table can have many
  records in a separate table.
• A foreign key is a field for which every entry in one table has a unique value in another table and
  where the field in the other table is the primary key for that table.
• The foreign key helps maintain the Rule of Referential Integrity: Every foreign key field value
  must appear in another table’s primary key field. Foreign keys enable information from multiple
  tables to be joined together for analysis purposes. There is a one-to-many relationship between a
  primary key and its corresponding foreign key.
• The simplest format of a SELECT query is

  SELECT * FROM tableName

  where the asterisk (*) indicates that all rows and columns from tableName should be selected and
  tableName specifies the table in the database from which the data will be selected.
• To select specific fields from a table, replace the asterisk (*) with a comma-separated list of the
  field names to select.
• Programmers process result sets by knowing in advance the order of the fields in the result set.
  Specifying the field names to select guarantees that the fields are always returned in the specified
  order, even if the actual order of the fields in the database table(s) changes.
• The optional WHERE clause in a SELECT query specifies the selection criteria for the query. The
  simplest format of a SELECT query with selection criteria is

  SELECT fieldName1, fieldName2, ... FROM tableName WHERE criteria

  where the WHERE clause condition can contain operators <, >, <=, >=, =, <> and LIKE. Operator LIKE
  is used for pattern matching with wildcard characters percent (%) and underscore (_).
• A percent character (%) in a pattern indicates that a string matching the pattern can have zero or
  more characters at the percent character’s location in the pattern.
• An underscore (_) in the pattern string indicates a single character at that position in the pattern.
• The results of a query can be arranged in ascending or descending order using the optional ORDER
  BY clause. The simplest form of an ORDER BY clause is

  SELECT fieldName1, fieldName2, ... FROM tableName ORDER BY field ASC
  SELECT fieldName1, fieldName2, ... FROM tableName ORDER BY field DESC

  where ASC specifies ascending order, DESC specifies descending order and field specifies the field
  on which the sort is based. The default sorting order is ascending, so ASC is optional.
• Multiple fields can be used for ordering purposes with an ORDER BY clause of the form

  ORDER BY field1 sortingOrder, field2 sortingOrder, ...

• The WHERE and ORDER BY clauses can be combined in one query.
• A join merges records from two or more tables by testing for matching values in a field that is com-
  mon to both tables. The simplest format of a join is

  SELECT fieldName1, fieldName2, ...
  FROM table1, table2
  WHERE table1.fieldName = table2.fieldName

  The query’s WHERE clause specifies the fields from each table that should be compared to deter-
  mine which records will be selected. These fields normally represent the primary key in one table
  and the corresponding foreign key in the other table.
• If an SQL statement uses fields with the same name from multiple tables, the field names must be fully qualified with its table name and a dot operator (.

• An **INSERT INTO** statement inserts a new record in a table. The simplest form of this statement is

```sql
INSERT INTO tablename ( fieldName1, fieldName2, ..., fieldNameN ) VALUES ( value1, value2, ..., valueN )
```

where **tableName** is the table in which to insert the record. The **tableName** is followed by a comma-separated list of field names in parentheses. The list of field names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses.

• SQL statements use single quote (') as a delimiter for strings. To specify a string containing a single quote in an SQL statement, the single quote must be escaped with another single quote.

• An **UPDATE** statement modifies data in a table. The simplest form for an **UPDATE** statement is

```sql
UPDATE tableName
SET fieldName1 = value1, fieldName2 = value2, ..., fieldNameN = valueN
WHERE criteria
```

where **tableName** is the table in which to update a record (or records). The **tableName** is followed by keyword **SET** and a comma-separated list of field name/value pairs in the format **fieldName = value**. The **WHERE** clause **criteria** determines the record(s) to update.

• A **DELETE** statement removes data from a table. The simplest form for a **DELETE** statement is

```sql
DELETE FROM tableName WHERE criteria
```

where **tableName** is the table from which to delete a record (or records). The **WHERE criteria** determines which record(s) to delete.

• Package **java.sql** contains classes and interfaces for manipulating relational databases in Java.

• A program must load the database driver class before the program can connect to the database.

• JDBC supports four categories of drivers: JDBC-to-ODBC bridge driver (Type 1); Native-API, partly Java driver (Type 2); JDBC-Net pure Java driver (Type 3) and Native-Protocol pure Java driver (Type 4). Type 3 and 4 drivers are preferred, because they are pure Java solutions.

• An object that implements interface **Connection** manages the connection between the Java program and the database. **Connection** objects enable programs to create SQL statements that manipulate databases and to perform transaction processing.

• Method **getConnection** of class **DriverManager** attempts to connect to a database specified by its URL argument. The URL helps the program locate the database. The URL includes the protocol for communication, the subprotocol for communication and the name of the database.

• **Connection** method **createStatement** creates an object of type **Statement**. The program uses the **Statement** object to submit SQL statements to the database.

• **Statement** method **executeQuery** executes a query that selects information from a table or set of tables and returns an object that implements interface **ResultSet** containing the query results. **ResultSet** methods enable a program to manipulate query results.

• A **ResultSetMetaData** object describes a **ResultSet**’s contents. Programs can use metadata programmatically to obtain information about the **ResultSet** column names and types.

• **ResultSetMetaData** method **getColumnCount** retrieves the number of columns in the **ResultSet**.

• **ResultSet** method **next** positions the **ResultSet** cursor to the next record in the **ResultSet**. The cursor keeps track of the current record. Method **next** returns **boolean** value **true**
if it is able to position to the next record; otherwise, the method returns \texttt{false}. This method must be called to begin processing a \texttt{ResultSet}.

- When processing \texttt{ResultSet}s, it is possible to extract each column of the \texttt{ResultSet} as a specific Java data type. \texttt{ResultSetMetaData} method \texttt{getColumnType} returns a constant integer from class \texttt{Types} (package \texttt{java.sql}) indicating the type of the data for a specific column.

- \texttt{ResultSet} \texttt{get} methods typically receive as an argument either a column number (as an \texttt{int}) or a column name (as a \texttt{String}) indicating which column’s value to obtain.

- \texttt{ResultSet} column numbers start at 1.

- Each \texttt{Statement} object can open only one \texttt{ResultSet} object at a time. When a \texttt{Statement} returns a new \texttt{ResultSet}, the \texttt{Statement} closes the prior \texttt{ResultSet}.

- \texttt{Connection} method \texttt{createStatement} has an overloaded version that takes two arguments: the result set type and the result set concurrency. The result set type specifies whether the \texttt{ResultSet}’s cursor is able to scroll in both directions or forward only and whether the \texttt{ResultSet} is sensitive to changes. The result set concurrency specifies whether the \texttt{ResultSet} can be updated with \texttt{ResultSet}’s update methods.

- Some \texttt{ResultSet} implementations do not support scrollable and/or updatable \texttt{ResultSets}.

- \texttt{TableModel} method \texttt{getColumnClass} returns a \texttt{Class} object that represents the superclass of all objects in a particular column. \texttt{JTable} uses this information to set up the default cell renderer and cell editor for that column in a \texttt{JTable}.

- \texttt{ResultSetMetaData} method \texttt{getColumnClassName} obtains the fully qualified class name of the specified column.

- \texttt{TableModel} method \texttt{getColumnCount} returns the number of columns in the model’s underlying \texttt{ResultSet}.

- \texttt{ResultSetMetaData} method \texttt{getColumnCount} obtains the number of columns in the \texttt{ResultSet}.

- \texttt{TableModel} method \texttt{getColumnName} returns the name of the column in the model’s underlying \texttt{ResultSet}.

- \texttt{ResultSetMetaData} method \texttt{getColumnName} obtains the column name from the \texttt{ResultSet}.

- \texttt{TableModel} method \texttt{getRowCount} returns the number of rows in the model’s underlying \texttt{ResultSet}.

- \texttt{TableModel} method \texttt{getValueAt} returns the \texttt{Object} in a particular row and column of the model’s underlying \texttt{ResultSet}.

- \texttt{ResultSet} method \texttt{absolute} positions the \texttt{ResultSet} cursor at a specific row.

- \texttt{AbstractTableModel} method \texttt{fireTableStructureChanged} notifies any \texttt{JTable} using a particular \texttt{TableModel} object as its model that the data in the model has changed.

- Interface \texttt{PreparedStatement} enables an application programmer to create SQL statements that are maintained in a compiled form that enables the statements to execute more efficiently than \texttt{Statement} objects. \texttt{PreparedStatement} objects are more flexible than \texttt{Statement} objects, because they can specify parameters.

- Question marks (?) in the SQL of a \texttt{PreparedStatement} represent placeholders for values that will be passed as part of the SQL statement to the database. Before the program executes a \texttt{PreparedStatement}, the program must specify the values of those parameters by using interface \texttt{PreparedStatement}’s \texttt{set} methods.

- Transaction processing enables a program that interacts with a database to treat a database operation (or set of operations) as a transaction. When the transaction completes, a decision can be made
to either commit the transaction or roll back the transaction. Java provides transaction processing via methods of interface `Connection`.

- Method `setAutoCommit` specifies whether each SQL statement commits after it completes (a `true` argument) or if SQL statements should be grouped as a transaction (a `false` argument).
- If autocommit is disabled the program must follow the last SQL statement in the transaction with a call to `Connection` method `commit` or `rollback`.
- JDBC enables programs to invoke stored procedures using objects that implement interface `CallableStatement`.

- `CallableStatement`s can receive arguments specified with the methods inherited from interface `PreparedStatement`. In addition, `CallableStatement`s can specify output parameters in which a stored procedure can place return values.
- A series of database updates can be performed in a batch update to the database. JDBC `Statement`s, `PreparedStatement`s and `CallableStatement`s provide an `addBatch` method that enables the program to add SQL statements to a batch for future execution. Each `Statement`, `PreparedStatement` or `CallableStatement` object maintains its own list of SQL statements to perform in a batch update.
- `Statement` method `executeBatch` executes the SQL statements in a batch and returns an array of `int` values containing the status of each SQL statement. If the database connection is in autocommit mode, the database commits each statement as it completes execution. Otherwise, `Connection` methods `commit` or `rollback` must be called as appropriate.
- Programs use `Statement` method `execute` to execute `Statement`s, `PreparedStatement`s and `CallableStatement`s that return multiple `ResultSet`s or update counts. Method `execute` returns a `boolean` indicating whether the first result is a `ResultSet` (`true`) or an update count (`false`). The program invokes method `getResultSet` or method `getUpdateCount` to obtain the first result and obtains subsequent results with `getMoreResults`.
- Some JDBC drivers support updatable `ResultSet`s. Such `ResultSet`s enable a program to insert, update and delete records using methods of interface `ResultSet`.
- Interface `ResultSet` provides `update` methods that enable the program to specify new values for particular columns in the current `ResultSet` row.
- Interface `ResultSet` provides methods `deleteRow`, `insertRow` and `updateRow` to manipulate the `ResultSet` and the underlying database.
- Every updatable `ResultSet` maintains an insert row where the program can build a new record before inserting it into the `ResultSet` and the database. Before invoking `ResultSet`’s `update` methods to build the new record, the program must invoke `ResultSet` method `moveToInsertRow`. The `ResultSet` keeps track of the cursor’s location before that operation and can return to the cursor location to continue processing the `ResultSet` by invoking `ResultSet` method `moveToCurrentRow`.
- A `DataSource` is new way for programs to obtain database connections that access databases which are external to those applications.
- Applications can establish connection pools that maintain many database connections. Databases that provide full support for the JDBC optional package include implementations of interfaces `ConnectionPoolDataSource` and `PooledConnection` for this purpose.
- The JDBC optional package introduces a new interface, `RowSet`, for manipulating tabular data sources such as `ResultSet`s. `RowSets` are not implemented as part of the database driver. Instead, they are implemented as JavaBeans that encapsulate a tabular data source.
• Interface `RowSet` extends interface `ResultSet`. Thus, a `RowSet` object has all the functionality of a `ResultSet`, including the ability to scroll through the records, insert new records, update existing records and delete existing records.

• Unlike `ResultSet`s, `RowSet`s implementations can be serializable so they can be saved locally or transmitted across a network.

• `RowSet`s support JavaBean events that enable an application using a `RowSet` to be notified when the `RowSet` cursor is moved, a record is inserted, a record is updated, a record is deleted or the entire set of data in the `RowSet` changes.

• Class `CachedRowSet` defines a disconnected `RowSet` that can be serialized. `CachedRowSet` provides full support for scrolling through data and updating data.

• `WebRowSet` is a subclass of `CachedRowSet` that enables `RowSet` data to be output as an XML document.

• Class `JDBCRowSet` defines a connected `RowSet` that encapsulates a `ResultSet` to make the `ResultSet` appear like a JavaBean to the program.

**TERMINOLOGY**

- SQL wildcard character
- _ SQL wildcard character
- `absolute` method of `ResultSet`
- `AbstractTableModel` class
- `addBatch` method of `PreparedStatement`
- `addBatch` method of `Statement`
- `addTableModelListener` method of `TableModel`
- `autocomplete` state
- `batch processing`
- `BatchUpdateException` class
- `CachedRowSet` class
- `CallableStatement` interface
- `clearBatch` method of `Statement`
- `close` method of `Connection`
- `close` method of `Statement`
- `Cloudscape database`
- `COM.cloudscape.core.RmiJdbcDriver`
- `commit a transaction`
- `commit` method of `Connection`
- `connect to a database`
- `Connection` interface
- `connection pool`
- `ConnectionPoolDataSource` interface
- `createStatement` method of `Connection`
- `database`
- `database driver`
- `DataSource` interface
- `DELETE FROM` SQL statement
- `deleteRow` method of `ResultSet`
- `disconnected RowSet`
- `DriverManager` class
- `execute` method of `Statement`
- `executeBatch` method of `Statement`
- `executeQuery` method of `Statement`
- `executeUpdate` method of `Statement`
- `field`
- `fireTableStructureChanged` method of `AbstractTableModel`
- `foreign key`
- `getAutoCommit` method of `Connection`
- `getColumnClass` method of `TableModel`
- `getColumnName` method of `ResultSetMetaData`
- `getUpdateCount` method of `Statement`
- `getUpdateCounts` method of `BatchUpdateException`
Self-Review Exercises

8.1 Fill in the blanks in each of the following statements:

a) The most popular database query language is _________.

b) A table in a database consists of ________ and _________.

c) Tables are manipulated in Java as ________ objects.

d) The ________ uniquely identifies each record in a table.

e) SQL keyword ________ is followed by the selection criteria that specify the records to select in a query.

f) SQL keywords ________ specify the order in which records are sorted in a query.

g) Selecting data from multiple database tables is called ________ the data.

h) A ________ is an integrated collection of data that is centrally controlled.

i) A ________ is a field in a table for which every entry has a unique value in another table and where the field in the other table is the primary key for that table.

j) Package ________ contains classes and interfaces for manipulating relational databases in Java.
k) Interface __________ helps manage the connection between the Java program and the database.

l) A __________ object is used to submit a query to a database.

**ANSWERS TO SELF-REVIEW EXERCISES**

8.1 a) SQL. b) rows, columns. c) ResultSet. d) primary key. e) WHERE. f) ORDER BY. g) joining. h) database. i) foreign key. j) java.sql. k) Connection. l) Statement.

**EXERCISES**

8.2 Using the techniques shown in this chapter, define a complete query application for the books database. Provide a series of predefined queries, with an appropriate name for each query, displayed in a JComboBox. Also allow users to supply their own queries and add them to the JComboBox. Provide the following predefined queries:

a) Select all authors from the Authors table.
b) Select all publishers from the Publishers table.
c) Select a specific author and list all books for that author. Include the title, year and ISBN number. Order the information alphabetically by the author’s last name and first name.
d) Select a specific publisher and list all books published by that publisher. Include the title, year and ISBN number. Order the information alphabetically by title.
e) Provide any other queries you feel are appropriate.

8.3 Modify Exercise 8.2 to define a complete database manipulation application for the books database. In addition to the querying, the user should be able to edit existing data and add new data to the database (obeying referential and entity integrity constraints). Allow the user to edit the database in the following ways:

a) Add a new author.
b) Edit the existing information for an author.
c) Add a new title for an author. (Remember that the book must have an entry in the AuthorISBN table.) Be sure to specify the publisher of the title.
d) Add a new publisher.
e) Edit the existing information for a publisher.
f) For each of the preceding database manipulations, design an appropriate GUI to allow the user to perform the data manipulation.

8.4 Modify the Search capability in the address book example of Fig. 8.33–Fig. 8.38 to allow the user to scroll through the ResultSet in case there is more than one person with the specified last name in the addressbook database. Provide an appropriate GUI.

8.5 Modify the address book example of Fig. 8.33–Fig. 8.38 to enable each address book entry to have multiple addresses, phone numbers and e-mail addresses. The user of the program should be able to view multiple addresses, phone numbers and e-mail addresses. The user also should be able to add, update or delete individual addresses, phone numbers and e-mail addresses. [Note: This exercise is large and requires substantial modifications to the original classes in the address book example.]

**BIBLIOGRAPHY**


Objectives

- To execute servlets with the Apache Tomcat server.
- To be able to respond to HTTP requests from an HttpServlet.
- To be able to redirect requests to static and dynamic Web resources.
- To be able to maintain session information with cookies and HttpSession objects.
- To be able to access a database from a servlet.

A fair request should be followed by the deed in silence.
Dante Alighieri

The longest part of the journey is said to be the passing of the gate.
Marcus Terentius Varro

If nominated, I will not accept; if elected, I will not serve.
General William T. Sherman

Me want cookie!
The Cookie Monster, Sesame Street

When to the sessions of sweet silent thought
I summon up remembrance of things past, …
William Shakespeare

Friends share all things.
Pythagorus

If at first you don’t succeed, destroy all evidence that you tried.
Newt Heilscher
### Outline

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>9.2</td>
<td>Servlet Overview and Architecture</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Interface Servlet and the Servlet Life Cycle</td>
</tr>
<tr>
<td>9.2.2</td>
<td>HttpServlet Class</td>
</tr>
<tr>
<td>9.2.3</td>
<td>HttpServletRequest Interface</td>
</tr>
<tr>
<td>9.2.4</td>
<td>HttpServletResponse Interface</td>
</tr>
<tr>
<td>9.3</td>
<td>Handling HTTP get Requests</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Setting Up the Apache Tomcat Server</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Deploying a Web Application</td>
</tr>
<tr>
<td>9.4</td>
<td>Handling HTTP get Requests Containing Data</td>
</tr>
<tr>
<td>9.5</td>
<td>Handling HTTP post Requests</td>
</tr>
<tr>
<td>9.6</td>
<td>Redirecting Requests to Other Resources</td>
</tr>
<tr>
<td>9.7</td>
<td>Session Tracking</td>
</tr>
<tr>
<td>9.7.1</td>
<td>Cookies</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Session Tracking with HttpSession</td>
</tr>
<tr>
<td>9.8</td>
<td>Multi-tier Applications: Using JDBC from a Servlet</td>
</tr>
<tr>
<td>9.9</td>
<td>HttpUtils Class</td>
</tr>
<tr>
<td>9.10</td>
<td>Internet and World Wide Web Resources</td>
</tr>
</tbody>
</table>

**Summary** • **Terminology** • **Self-Review Exercises** • **Answers to Self-Review Exercises** • **Exercises**

### 9.1 Introduction

There is much excitement over the Internet and the World Wide Web. The Internet ties the “information world” together. The World Wide Web makes the Internet easy to use and gives it the flair and sizzle of multimedia. Organizations see the Internet and the Web as crucial to their information systems strategies. Java provides a number of built-in networking capabilities that make it easy to develop Internet-based and Web-based applications. Not only can Java specify parallelism through multithreading, but it can enable programs to search the world for information and to collaborate with programs running on other computers internationally, nationally or just within an organization. Java can even enable applets and applications running on the same computer to communicate with one another, subject to security constraints.

Networking is a massive and complex topic. Computer science and computer engineering students typically take a full-semester, upper-level course in computer networking and continue with further study at the graduate level. Java provides a rich complement of networking capabilities and will likely be used as an implementation vehicle in computer networking courses. In Advanced Java 2 Platform How to Program we introduce several Java networking concepts and capabilities.

Java’s networking capabilities are grouped into several packages. The fundamental networking capabilities are defined by classes and interfaces of package `java.net`. 
through which Java offers *socket-based communications* that enable applications to view networking as streams of data—a program can read from a *socket* or write to a socket as simply as reading from a file or writing to a file. The classes and interfaces of package `java.net` also offer *packet-based communications* that enable individual *packets* of information to be transmitted—commonly used to transmit audio and video over the Internet. Our book *Java How to Program, Fourth Edition* shows how to create and manipulate sockets and how to communicate with packets of data.

Higher-level views of networking are provided by classes and interfaces in the `java.rmi` packages (five packages) for *Remote Method Invocation (RMI)* and `org.omg` packages (seven packages) for *Common Object Request Broker Architecture (CORBA)* that are part of the Java 2 API. The RMI packages allow Java objects running on separate Java Virtual Machines (normally on separate computers) to communicate via remote method calls. Such method calls appear to be to an object in the same program, but actually have built-in networking (based on the capabilities of package `java.net`) that communicates the method calls to another object on a separate computer. The CORBA packages provide similar functionality to the RMI packages. A key difference between RMI and CORBA is that RMI can only be used between Java objects, whereas CORBA can be used between any two applications that understand CORBA—including applications written in other programming languages. In Chapter 13 of *Advanced Java 2 Platform How to Program*, we present Java’s RMI capabilities. Chapters 26–27 of *Advanced Java 2 Platform How to Program* discuss the basic CORBA concepts and present a case study that implements a distributed system in CORBA.

Our discussion of networking over the next two chapters focuses on both sides of a *client-server relationship*. The *client* requests that some action be performed and the *server* performs the action and responds to the client. This request-response model of communication is the foundation for the highest-level views of networking in Java—*servlets* and *Java-Server Pages (JSP)*. A servlet extends the functionality of a server. Packages `javax.servlet` and `javax.servlet.http` provide the classes and interfaces to define servlets. Packages `javax.servlet.jsp` and `javax.servlet.jsp.tagext` provide the classes and interfaces that extend the servlet capabilities for JavaServer Pages. Using special syntax, JSP allows Web-page implementors to create pages that use encapsulated Java functionality and even to write *scriptlets* of actual Java code directly in the page.

A common implementation of the request-response model is between World Wide Web browsers and World Wide Web servers. When a user selects a Web site to browse through their browser (the client application), a request is sent to the appropriate Web server (the server application). The server normally responds to the client by sending the appropriate XHTML Web page. Servlets are effective for developing Web-based solutions that help provide secure access to a Web site, interact with databases on behalf of a client, dynamically generate custom XHTML documents to be displayed by browsers and maintain unique session information for each client.

**Software Engineering Observation 9.1**

Although servlets typically are used in distributed Web applications, not all servlets are required to enhance the functionality of a Web server.

This chapter begins our networking discussions with servlets that enhance the functionality of World Wide Web servers—the most common form of servlet today. Chapter 10 discusses JSPs, which are translated into servlets. JSPs are a convenient and powerful way
to implement the request/response mechanism of the Web without getting into the lower-level details of servlets. Together, servlets and JSPs form the Web tier of the Java 2 Enterprise Edition (J2EE).

Many developers feel that servlets are the right solution for database-intensive applications that communicate with so-called thin clients—applications that require minimal client-side support. The server is responsible for database access. Clients connect to the server using standard protocols available on most client platforms. Thus, the presentation-logic code for generating dynamic content can be written once and reside on the server for access by clients, to allow programmers to create efficient thin clients.

In this chapter, our servlet examples demonstrate the Web’s request/response mechanism (primarily with get and post requests), session-tracking capabilities, redirecting requests to other resources and interacting with databases through JDBC. We placed this chapter after our discussion of JDBC and databases intentionally, so that we can build multi-tier, client–server applications that access databases. In Chapter 11, we build a bookstore Web application, using XML technologies (Appendices A-D), JDBC technology from Chapter 8, the servlet technology from this chapter and the JSP technology from Chapter 10. We present additional servlet capabilities in the case study.

Sun Microsystems, through the Java Community Process, is responsible for the development of the servlet and JavaServer Pages specifications. The reference implementation of both these standards is under development by the Apache Software Foundation (www.apache.org) as part of the Jakarta Project (jakarta.apache.org). As stated on the Jakarta Project’s home page, “The goal of the Jakarta Project is to provide commercial-quality server solutions based on the Java Platform that are developed in an open and cooperative fashion.” There are many subprojects under the Jakarta project to help commercial server-side developers. The servlet and JSP part of the Jakarta Project is called Tomcat. This is the official reference implementation of the JSP and servlet standards. We use Tomcat to demonstrate the servlets in this chapter. The most recent implementation of Tomcat at the time of this writing was version 3.2.3. For your convenience, Tomcat 3.2.3 is included on the CD that accompanies Advanced Java 2 Platform How to Program. However, the most recent version always can be downloaded from the Apache Group’s Web site. To execute the servlets in this chapter, you must install Tomcat or an equivalent servlet and JavaServer Pages implementation. We discuss the set up and configuration of Tomcat in Section 9.3.1 and Section 9.3.2 after we introduce our first example.

In our directions for testing each of the examples in this chapter, we indicate that you should copy files into specific Tomcat directories. All the example files for this chapter are located on the CD that accompanies this book and on our Web site www.deitel.com.

[Note: At the end of Section 9.10, we provide a list of Internet specifications (as discussed in the Servlet 2.2 Specification) for technologies related to servlet development. Each is listed with its RFC (Request for Comments) number. We provide the URL of a Web site that allows you to locate each specification for your review.]

### 9.2 Servlet Overview and Architecture

In this section, we overview Java servlet technology. We discuss at a high level the servlet-related classes, methods and exceptions. The next several sections present live-code examples in which we build multi-tier client–server systems using servlet and JDBC technology.
The Internet offers many protocols. The HTTP (Hypertext Transfer Protocol) that forms the basis of the World Wide Web uses URIs (Uniform Resource Identifiers—sometimes called Universal Resource Locators or URLs) to locate resources on the Internet. Common URIs represent files or directories and can represent complex tasks such as database lookups and Internet searches. For more information on URL formats, visit www.w3.org/Addressing

For more information on the HTTP protocol, visit www.w3.org/Protocols/HTTP

For information on a variety of World Wide Web topics, visit www.w3.org

JavaServer Pages technology is an extension of servlet technology. Normally, JSPs are used primarily when most of the content sent to the client is static text and markup, and only a small portion of the content is generated dynamically with Java code. Normally, servlets are used when a small portion of the content sent to the client is static text or markup. In fact, some servlets do not produce content. Rather, they perform a task on behalf of the client, then invoke other servlets or JSPs to provide a response. Note that in most cases servlet and JSP technologies are interchangeable. The server that executes a servlet often is referred to as the *servlet container* or *servlet engine*.

Servlets and JavaServer Pages have become so popular that they are now supported directly or with third-party plug-ins by most major Web servers and application servers, including the Netscape iPlanet Application Server, Microsoft’s Internet Information Server (IIS), the Apache HTTP Server, BEA’s WebLogic application server, IBM’s WebSphere application server, the World Wide Web Consortium’s Jigsaw Web server, and many more.

The servlets in this chapter demonstrate communication between clients and servers via the HTTP protocol. A client sends an HTTP request to the server or servlet container. The server or servlet container receives the request and directs it to be processed by the appropriate servlet. The servlet does its processing, which may include interacting with a database or other server-side components such as other servlets, JSPs or Enterprise JavaBeans (Chapter 16). The servlet returns its results to the client—normally in the form of an HTML, XHTML or XML document to display in a browser, but other data formats, such as images and binary data, can be returned.

### 9.2.1 Interface Servlet and the Servlet Life Cycle

Architecturally, all servlets must implement the *Servlet* interface. As with many key applet methods, the methods of interface *Servlet* are invoked automatically (by the server on which the servlet is installed, also known as the servlet container). This interface defines five methods described in Fig. 9.1.

Software Engineering Observation 9.2

All servlets must implement the *Servlet* interface of package *javax.servlet*.
Chapter 9 Servlets 535

A servlet’s life cycle begins when the servlet container loads the servlet into memory—normally, in response to the first request that the servlet receives. Before the servlet can handle that request, the servlet container invokes the servlet’s `init` method.

After `init` completes execution, the servlet can respond to its first request. All requests are handled by a servlet’s `service` method, which receives the request, processes the request and sends a response to the client. During a servlet’s life cycle, method `service` is called once per request. Each new request typically results in a new thread of execution (created by the servlet container) in which method `service` executes. When the servlet container terminates the servlet, the servlet’s `destroy` method is called to release servlet resources.

**Performance Tip 9.1**

Starting a new thread for each request is more efficient than starting an entirely new process, as is the case in some other server-side technologies such as CGI. [Note: Like servlets, Fast CGI eliminates the overhead of starting a new process for each request.]

The servlet packages define two abstract classes that implement the interface `Servlet`—class `GenericServlet` (from the package `javax.servlet`) and class `HttpServlet` (from the package `javax.servlet.http`). These classes provide default implementations of all the `Servlet` methods. Most servlets extend either `GenericServlet` or `HttpServlet` and override some or all of their methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void init(ServletConfig config)</code></td>
<td>This method is automatically called once during a servlet’s execution cycle to initialize the servlet. The <code>ServletConfig</code> argument is supplied by the servlet container that executes the servlet.</td>
</tr>
<tr>
<td><code>ServletConfig getServletConfig()</code></td>
<td>This method returns a reference to an object that implements interface <code>ServletConfig</code>. This object provides access to the servlet’s configuration information such as servlet initialization parameters and the servlet’s <code>ServletContext</code>, which provides the servlet with access to its environment (i.e., the servlet container in which the servlet executes).</td>
</tr>
<tr>
<td><code>String getServletInfo()</code></td>
<td>This method is defined by a servlet programmer to return a <code>String</code> containing servlet information such as the servlet’s author and version.</td>
</tr>
<tr>
<td><code>void service(ServletRequest request, ServletResponse response)</code></td>
<td>The servlet container calls this method to respond to a client request to the servlet.</td>
</tr>
<tr>
<td><code>void destroy()</code></td>
<td>This “cleanup” method is called when a servlet is terminated by its servlet container. Resources used by the servlet, such as an open file or an open database connection, should be deallocated here.</td>
</tr>
</tbody>
</table>
The examples in this chapter all extend class `HttpServlet`, which defines enhanced processing capabilities for servlets that extend the functionality of a Web server. The key method in every servlet is `service`, which receives both a `ServletRequest` object and a `ServletResponse` object. These objects provide access to input and output streams that allow the servlet to read data from the client and send data to the client. These streams can be either byte based or character based. If problems occur during the execution of a servlet, either `ServletExceptions` or `IOExceptions` are thrown to indicate the problem.

**Software Engineering Observation 9.3**

Servlets can implement tagging interface `javax.servlet.SingleThreadModel` to indicate that only one thread of execution may enter method `service` on a particular servlet instance at a time. When a servlet implements `SingleThreadModel`, the servlet container can create multiple instances of the servlet to handle multiple requests to the servlet in parallel. In this case, you may need to provide synchronized access to shared resources used by method `service`.

### 9.2.2 `HttpServletRequest` Class

Web-based servlets typically extend class `HttpServletRequest`. Class `HttpServletRequest` over-rides method `service` to distinguish between the typical requests received from a client Web browser. The two most common `HTTP request types` (also known as `request methods`) are `get` and `post`. A `get` request gets (or retrieves) information from a server. Common uses of `get` requests are to retrieve an HTML document or an image. A `post` request posts (or sends) data to a server. Common uses of `post` requests typically send information, such as authentication information or data from a `form` that obtains user input, to a server.

Class `HttpServletRequest` defines methods `doGet` and `doPost` to respond to `get` and `post` requests from a client, respectively. These methods are called by the `service` method, which is called when a request arrives at the server. Method `service` first determines the request type, then calls the appropriate method for handling such a request. Other less common request types are beyond the scope of this book. Methods of class `HttpServletRequest` that respond to the other request types are shown in Fig. 9.2. They all receive parameters of type `HttpServletRequest` and `HttpServletResponse` and return `void`. The methods of Fig. 9.2 are not frequently used. For more information on the HTTP protocol, visit

[www.w3.org/Protocols](http://www.w3.org/Protocols).

**Software Engineering Observation 9.4**

Do not override method `service` in an `HttpServletRequest` subclass. Doing so prevents the servlet from distinguishing between request types.

Methods `doGet` and `doPost` receive as arguments an `HttpServletRequest` object and an `HttpServletResponse` object that enable interaction between the client and the server. The methods of `HttpServletRequest` make it easy to access the data supplied as part of the request. The `HttpServletResponse` methods make it easy to return the servlet’s results to the Web client. Interfaces `HttpServletRequest` and `HttpServletResponse` are discussed in the next two sections.
9.2.3 HttpServletRequest Interface

Every call to `doGet` or `doPost` for an `HttpServlet` receives an object that implements interface `HttpServletRequest`. The Web server that executes the servlet creates an `HttpServletRequest` object and passes this to the servlet’s `service` method (which, in turn, passes it to `doGet` or `doPost`). This object contains the request from the client. A variety of methods are provided to enable the servlet to process the client’s request. Some of these methods are from interface `ServletRequest`—the interface that `HttpServletRequest` extends. A few key methods used in this chapter are presented in Fig. 9.3. You can view a complete list of `HttpServletRequest` methods online at

```
java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/http/
HttpServletRequest.html
```

or you can download and install Tomcat (discussed in Section 9.3.1) and view the documentation on your local computer.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doDelete</code></td>
<td>Called in response to an HTTP <code>delete</code> request. Such a request is normally used to delete a file from a server. This may not be available on some servers, because of its inherent security risks (i.e., the client could delete a file that is critical to the execution of the server or an application).</td>
</tr>
<tr>
<td><code>doOptions</code></td>
<td>Called in response to an HTTP <code>options</code> request. This returns information to the client indicating the HTTP options supported by the server, such as the version of HTTP (1.0 or 1.1) and the request methods the server supports.</td>
</tr>
<tr>
<td><code>doPut</code></td>
<td>Called in response to an HTTP <code>put</code> request. Such a request is normally used to store a file on the server. This may not be available on some servers, because of its inherent security risks (i.e., the client could place an executable application on the server, which, if executed, could damage the server—perhaps by deleting critical files or occupying resources).</td>
</tr>
<tr>
<td><code>doTrace</code></td>
<td>Called in response to an HTTP <code>trace</code> request. Such a request is normally used for debugging. The implementation of this method automatically returns an HTML document to the client containing the request header information (data sent by the browser as part of the request).</td>
</tr>
</tbody>
</table>

**Fig. 9.3** Some methods of interface `HttpServletRequest` (part 1 of 2).

```java
String getParameter( String name )
```

Obtains the value of a parameter sent to the servlet as part of a `get` or `post` request. The `name` argument represents the parameter name.
Every call to `doGet` or `doPost` for an `HttpServlet` receives an object that implements interface `HttpServletResponse`. The Web server that executes the servlet creates an `HttpServletResponse` object and passes it to the servlet’s `service` method (which, in turn, passes it to `doGet` or `doPost`). This object provides a variety of methods that enable the servlet to formulate the response to the client. Some of these methods are from interface `ServletRequest`—the interface that `HttpServletResponse` extends. A few key methods used in this chapter are presented in Fig. 9.4. You can view a complete list of `HttpServletResponse` methods online at


or you can download and install Tomcat (discussed in Section 9.3.1) and view the documentation on your local computer..

### Fig. 9.4 Some methods of interface `HttpServletResponse` (part 1 of 2).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Enumeration getParameterNames()</code></td>
<td>Returns the names of all the parameters sent to the servlet as part of a <code>post</code> request.</td>
</tr>
<tr>
<td><code>String[] getParameterValues( String name )</code></td>
<td>For a parameter with multiple values, this method returns an array of <code>Strings</code> containing the values for a specified servlet parameter.</td>
</tr>
<tr>
<td><code>Cookie[] getCookies()</code></td>
<td>Returns an array of <code>Cookie</code> objects stored on the client by the server. <code>Cookies</code> can be used to uniquely identify clients to the servlet.</td>
</tr>
<tr>
<td><code>HttpSession getSession( boolean create )</code></td>
<td>Returns an <code>HttpSession</code> object associated with the client’s current browsing session. An <code>HttpSession</code> object can be created by this method (true argument) if an <code>HttpSession</code> object does not already exist for the client. <code>HttpSession</code> objects can be used in similar ways to <code>Cookies</code> for uniquely identifying clients.</td>
</tr>
</tbody>
</table>

### 9.2.4 `HttpServletResponse` Interface

Every call to `doGet` or `doPost` for an `HttpServlet` receives an object that implements interface `HttpServletResponse`. The Web server that executes the servlet creates an `HttpServletResponse` object and passes it to the servlet’s `service` method (which, in turn, passes it to `doGet` or `doPost`). This object provides a variety of methods that enable the servlet to formulate the response to the client. Some of these methods are from interface `ServletRequest`—the interface that `HttpServletResponse` extends. A few key methods used in this chapter are presented in Fig. 9.4. You can view a complete list of `HttpServletResponse` methods online at


or you can download and install Tomcat (discussed in Section 9.3.1) and view the documentation on your local computer..

### Fig. 9.4 Some methods of interface `HttpServletResponse` (part 1 of 2).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void addCookie( Cookie cookie )</code></td>
<td>Used to add a <code>Cookie</code> to the header of the response to the client. The <code>Cookie</code>'s maximum age and whether <code>Cookies</code> are enabled on the client determine if <code>Cookies</code> are stored on the client.</td>
</tr>
</tbody>
</table>
9.3 Handling HTTP get Requests

The primary purpose of an HTTP get request is to retrieve the content of a specified URL—normally the content is an HTML or XHTML document (i.e., a Web page). The servlet of Fig. 9.5 and the XHTML document of Fig. 9.6 demonstrate a servlet that handles HTTP get requests. When the user clicks the Get HTML Document button (Fig. 9.6), a get request is sent to the servlet WelcomeServlet (Fig. 9.5). The servlet responds to the request by generating dynamically an XHTML document for the client that displays “Welcome to Servlets!” Figure 9.5 shows the WelcomeServlet source code. Figure 9.6 shows the XHTML document the client loads to access the servlet and shows screen captures of the client’s browser window before and after the interaction with the servlet. [Note: Section 9.3.1 discusses how to set up and configure Tomcat to execute this example.]

Lines 5 and 6 import the javax.servlet and javax.servlet.http packages. We use several data types from these packages in the example.

Package javax.servlet.http provides superclass HttpServlet for servlets that handle HTTP get requests and HTTP post requests. This class implements interface javax.servlet.Servlet and adds methods that support HTTP protocol requests. Class WelcomeServlet extends HttpServlet (line 9) for this reason.

Superclass HttpServlet provides method doGet to respond to get requests. Its default functionality is to indicate a “Method not allowed” error. Typically, this error is indicated in Internet Explorer with a Web page that states “This page cannot be displayed” and in Netscape Navigator with a Web page that states “Error: 405.” Lines 12–44 override method doGet to provide custom get request processing. Method doGet receives two arguments—an HttpServletRequest object and an HttpServletResponse object (both from package javax.servlet.http). The HttpServletRequest object represents the client’s request, and the HttpServletResponse object represents the server’s response to the client. If method doGet is unable to handle a client’s request, it throws an exception of type javax.servlet.ServletException. If doGet encounters an error during stream processing (reading from the client or writing to the client), it throws a java.io.IOException.
To demonstrate a response to a get request, our servlet creates an XHTML document containing the text “Welcome to Servlets!”. The text of the XHTML document is the response to the client. The response is sent to the client through the PrintWriter object obtained from the HttpServletResponse object.
Line 16 uses the `response` object’s `setContent_type` method to specify the content type of the data to be sent as the response to the client. This enables the client browser to understand and handle the content. The content type also is known as the MIME type (Multipurpose Internet Mail Extension) of the data. In this example, the content type is `text/html` to indicate to the browser that the response is an XHTML document. The browser knows that it must read the XHTML tags in the document, format the document according to the tags and display the document in the browser window. For more information on MIME types visit [www.irvine.com/~mime](http://www.irvine.com/~mime).

Line 17 uses the `response` object’s `getWriter` method to obtain a reference to the `PrintWriter` object that enables the servlet to send content to the client. [Note: If the response is binary data, such as an image, method `getOutputStream` is used to obtain a reference to a `ServletOutputStream` object.]

Lines 22–42 create the XHTML document by writing strings with the `out` object’s `println` method. This method outputs a newline character after its `String` argument. When rendering the Web page, the browser does not use the newline character. Rather, the newline character appears in the XHTML source that you can see by selecting `Source` from the `View` menu in Internet Explorer or `Page Source` from the `View` menu in Netscape Navigator. Line 43 closes the output stream, flushes the output buffer and sends the information to the client. This commits the response to the client.

The XHTML document in Fig. 9.6 provides a form that invokes the servlet defined in Fig. 9.5. The form’s action `/advjhtp1/welcome` specifies the URL path that invokes the servlet, and the form’s method indicates that the browser sends a get request to the server, which results in a call to the servlet’s `doGet` method. The URL specified as the action in this example is discussed in detail in Section 9.3.2 after we show how to set up and configure the Apache Tomcat server to execute the servlet in Fig. 9.5.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.6: WelcomeServlet.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
  <title>Handling an HTTP Get Request</title>
</head>
<body>
<form action = "/advjhtp1/welcome1" method = "get">
  <p><label>Click the button to invoke the servlet
    <input type = "submit" value = "Get HTML Document" />
  </label></p>
</form>
</body>
</html>
```

Fig. 9.6 HTML document in which the form’s action invokes WelcomeServlet through the alias `welcome1` specified in `web.xml` (part 1 of 2).
Note that the sample screen captures show a URL containing the server name local-host—a well-known server host name on most computers that support TCP/IP-based networking protocols such as HTTP. We often use localhost to demonstrate networking programs on the local computer, so that readers without a network connection can still learn network programming concepts. In this example, localhost indicates that the server on which the servlet is installed is running on the local machine. The server host name is followed by :8080, specifying the TCP port number at which the Tomcat server awaits requests from clients. Web browsers assume TCP port 80 by default as the server port at which clients make requests, but the Tomcat server awaits client requests at TCP port 8080. This allows Tomcat to execute on the same computer as a standard Web server application without affecting the Web server application’s ability to handle requests. If we do not explicitly specify the port number in the URL, the servlet never will receive our request and an error message will be displayed in the browser.

Software Engineering Observation 9.5

The Tomcat documentation specifies how to integrate Tomcat with popular Web server applications such as the Apache HTTP Server and Microsoft’s IIS.

Ports in this case are not physical hardware ports to which you attach cables; rather, they are logical locations named with integer values that allow clients to request different services on the same server. The port number specifies the logical location where a server waits for and receives connections from clients—this is also called the handshake point. When a client connects to a server to request a service, the client must specify the port number for that service; otherwise, the client request cannot be processed. Port numbers are positive integers with values up to 65,535, and there are separate sets of these port numbers for both the TCP and UDP protocols. Many operating systems reserve port numbers below...
1024 for system services (such as email and World Wide Web servers). Generally, these ports should not be specified as connection ports in your own server programs. In fact, some operating systems require special access privileges to use port numbers below 1024.

With so many ports from which to choose, how does a client know which port to use when requesting a service? The term well-known port number often is used when describing popular services on the Internet such as Web servers and email servers. For example, a Web server waits for clients to make requests at port 80 by default. All Web browsers know this number as the well-known port on a Web server where requests for HTML documents are made. So when you type a URL into a Web browser, the browser normally connects to port 80 on the server. Similarly, the Tomcat server uses port 8080 as its port number. Thus, requests to Tomcat for Web pages or to invoke servlets and JavaServer Pages must specify that the Tomcat server waiting for requests on port 8080.

The client can access the servlet only if the servlet is installed on a server that can respond to servlet requests. In some cases, servlet support is built directly into the Web server, and no special configuration is required to handle servlet requests. In other cases, it is necessary to integrate a servlet container with a Web server (as can be done with Tomcat and the Apache or IIS Web servers). Web servers that support servlets normally have an installation procedure for servlets. If you intend to execute your servlet as part of a Web server, please refer to your Web server’s documentation on how to install a servlet. For our examples, we demonstrate servlets with the Apache Tomcat server. Section 9.3.1 discusses the setup and configuration of Tomcat for use with this chapter. Section 9.3.2 discusses the deployment of the servlet in Fig. 9.5.

9.3.1 Setting Up the Apache Tomcat Server

Tomcat is a fully functional implementation of the JSP and servlet standards. It includes a Web server, so it can be used as a standalone test container for JSPs and servlets. Tomcat also can be specified as the handler for JSP and servlet requests received by popular Web servers such as the Apache Software Foundation’s Apache Web server or Microsoft’s Internet Information Server (IIS). Tomcat is integrated into the Java 2 Enterprise Edition reference implementation from Sun Microsystems.

The most recent release of Tomcat (version 3.2.3) can be downloaded from

jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.3/bin/

where there are a number of archive files. The complete Tomcat implementation is contained in the files that begin with the name jakarta-tomcat-3.2.3. Zip, tar and compressed tar files are provided for Windows, Linux and Solaris.

Extract the contents of the archive file to a directory on your hard disk. By default, the name of the directory containing Tomcat is jakarta-tomcat-3.2.3. For Tomcat to work correctly, you must define environment variables JAVA_HOME and TOMCAT_HOME. JAVA_HOME should point to the directory containing your Java installation (ours is d:\jdk1.3.1), and TOMCAT_HOME should point to the directory that contains Tomcat (ours is d:\jakarta-tomcat-3.2.3).

Testing and Debugging Tip 9.1

On some platforms you may need to restart your computer for the new environment variables to take effect.
After setting the environment variables, you can start the Tomcat server. Open a command prompt (or shell) and change directories to `bin` in `jakarta-tomcat-3.2.3`. In this directory are the files `tomcat.bat` and `tomcat.sh`, for starting the Tomcat server on Windows and UNIX (Linux or Solaris), respectively. To start the server, type

```
tomcat start
```

This launches the Tomcat server. The Tomcat server executes on TCP port 8080 to prevent conflicts with standard Web servers that typically execute on TCP port 80. To prove that Tomcat is executing and can respond to requests, open your Web browser and enter the URL

```
http://localhost:8080/
```

This should display the Tomcat documentation home page (Fig. 9.7). The host `localhost` indicates to the Web browser that it should request the home page from the Tomcat server on the local computer.

If the Tomcat documentation home page does not display, try the URL

```
http://127.0.0.1:8080/
```

The host `localhost` translates to the IP address `127.0.0.1`.

Testing and Debugging Tip 9.2

If the host name `localhost` does not work on your computer, substitute the IP address `127.0.0.1` instead.

To shut down the Tomcat server, issue the command

```
tomcat stop
```

from a command prompt (or shell).

---

![Tomcat documentation home page](Image)

*Fig. 9.7 Tomcat documentation home page. (Courtesy of The Apache Software Foundation.)*
9.3.2 Deploying a Web Application

JSPs, servlets and their supporting files are deployed as part of Web applications. Normally, Web applications are deployed in the `webapps` subdirectory of `jakarta-tomcat-3.2.3`. A Web application has a well-known directory structure in which all the files that are part of the application reside. This directory structure can be created by the server administrator in the `webapps` directory, or the entire directory structure can be archived in a Web application archive file. Such an archive is known as a WAR file and ends with the `.war` file extension. If a WAR file is placed in the `webapps` directory, then, when the Tomcat server begins execution, it extracts the contents of the WAR file into the appropriate `webapps` subdirectory structure. For simplicity as we teach servlets and JavaServer Pages, we create the already expanded directory structure for all the examples in this chapter and Chapter 10.

The Web application directory structure contains a context root—the top-level directory for an entire Web application—and several subdirectories. These are described in Fig. 9.8.

**Common Programming Error 9.1**

Using “servlet” or “servlets” as a context root may prevent a servlet from working correctly on some servers.

Configuring the context root for a Web application in Tomcat simply requires creating a subdirectory in the `webapps` directory. When Tomcat begins execution, it creates a context root for each subdirectory of `webapps`, using each subdirectory’s name as a context root name. To test the examples in this chapter and Chapter 10, create the directory `advjhtp1` in Tomcat’s `webapps` directory.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context root</td>
<td>This is the root directory for the Web application. The name of this directory is chosen by the Web application developer. All the JSPs, HTML documents, servlets and supporting files such as images and class files reside in this directory or its subdirectories. The name of this directory is specified by the Web application creator. To provide structure in a Web application, subdirectories can be placed in the context root. For example, if your application uses many images, you might place an images subdirectory in this directory. The examples of this chapter and Chapter 10 use <code>advjhtp1</code> as the context root.</td>
</tr>
<tr>
<td>WEB-INF</td>
<td>This directory contains the Web application deployment descriptor (<code>web.xml</code>).</td>
</tr>
<tr>
<td>WEB-INF/classes</td>
<td>This directory contains the servlet class files and other supporting class files used in a Web application. If the classes are part of a package, the complete package directory structure would begin here.</td>
</tr>
<tr>
<td>WEB-INF/lib</td>
<td>This directory contains Java archive (JAR) files. The JAR files can contain servlet class files and other supporting class files used in a Web application.</td>
</tr>
</tbody>
</table>

Fig. 9.8  Web application standard directories.
After configuring the context root, we must configure our Web application to handle the requests. This configuration occurs in a deployment descriptor, which is stored in a file called `web.xml`. The deployment descriptor specifies various configuration parameters such as the name used to invoke the servlet (i.e., its alias), a description of the servlet, the servlet’s fully qualified class name and a servlet mapping (i.e., the path or paths that cause the servlet container to invoke the servlet). You must create the `web.xml` file for this example. Many Java Web-application deployment tools create the `web.xml` file for you. The `web.xml` file for the first example in this chapter is shown in Fig. 9.9. We enhance this file as we add other servlets to the Web application throughout this chapter.

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
 "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

<!-- General description of your Web application -->
<display-name>
  Advanced Java How to Program JSP
  and Servlet Chapter Examples
</display-name>

<description>
  This is the Web application in which we
demonstrate our JSP and Servlet examples.
</description>

<!-- Servlet definitions -->
< servlet>
  <servlet-name>welcome1</servlet-name>

  <description>
    A simple servlet that handles an HTTP get request.
  </description>

  <servlet-class>
    com.deitel.advjhtp1.servlets.WelcomeServlet
  </servlet-class>
</servlet>

<!-- Servlet mappings -->
<servlet-mapping>
  <servlet-name>welcome1</servlet-name>
  <url-pattern>/welcome1</url-pattern>
</servlet-mapping>

</web-app>
```

Fig. 9.9 Deployment descriptor (`web.xml`) for the `advjhtp1` Web application.
Lines 1–3 specify the document type for the Web application deployment descriptor and the location of the DTD for this XML file. Element **web-app** (lines 5–37) defines the configuration of each servlet in the Web application and the servlet mapping for each servlet. Element **display-name** (lines 8–11) specifies a name that can be displayed to the administrator of the server on which the Web application is installed. Element **description** (lines 13–16) specifies a description of the Web application that might be displayed to the administrator of the server.

Element **servlet** (lines 19–29) describes a servlet. Element **servlet-name** (line 20) is the name we chose for the servlet (**welcome1**). Element **description** (lines 22–24) specifies a description for this particular servlet. Again, this can be displayed to the administrator of the Web server. Element **servlet-class** (lines 26–28) specifies compiled servlet’s fully qualified class name. Thus, the servlet **welcome1** is defined by class **com.deitel.advjhtp1.servlets.WelcomeServlet**.

Element **servlet-mapping** (lines 32–35) specifies **servlet-name** and **url-pattern** elements. The **URL pattern** helps the server determine which requests are sent to the servlet (**welcome1**). Our Web application will be installed as part of the **advjhtp1** context root discussed in Section 9.3.2. Thus, the URL we supply to the browser to invoke the servlet in this example is

```
/advjhtp1/welcome1
```

where **/advjhtp1** specifies the context root that helps the server determine which Web application handles the request and **/welcome1** specifies the URL pattern that is mapped to servlet **welcome1** to handle the request. Note that the server on which the servlet resides is not specified here, although it is possible to do so as follows:

```
http://localhost:8080/advjhtp1/welcome1
```

If the explicit server and port number are not specified as part of the URL, the browser assumes that the form handler (i.e., the servlet specified in the **action** property of the **form** element) resides at the same server and port number from which the browser downloaded the Web page containing the **form**.

There are several URL pattern formats that can be used. The **/welcome1** URL pattern requires an exact match of the pattern. You can also specify **path mappings**, extension mappings and a **default servlet** for a Web application. A path mapping begins with a **/** and ends with a **/**. For example, the URL pattern

```
/advjhtp1/example/*
```

indicates that any URL path beginning with **/advjhtp1/example/** will be sent to the servlet that has the preceding URL pattern. An extension mapping begins with **.** and ends with a **.**. For example, the URL pattern

```
*.jsp
```

indicates that any request for a file with the extension **.jsp** will be sent to the servlet that handles JSP requests. In fact, servers with JSP containers have an implicit mapping of the **.jsp** extension to a servlet that handles JSP requests. The URL pattern **/** represents the default servlet for the Web application. This is similar to the default document of a Web
server. For example, if you type the URL `www.deitel.com` into your Web browser, the document you receive from our Web server is the default document `index.html`. If the URL pattern matches the default servlet for a Web application, that servlet is invoked to return a default response to the client. This can be useful for personalizing Web content to specific users. We discuss personalization in Section 9.7, Session Tracking.

Finally, we are ready to place our files into the appropriate directories to complete the deployment of our first servlet, so we can test it. There are three files we must place in the appropriate directories—`WelcomeServlet.html`, `WelcomeServlet.class` and `web.xml`. In the `webapps` subdirectory of your `jakarta-tomcat-3.2.3` directory, create the `advjhtpl` subdirectory that represents the context root for our Web application. In this directory, create subdirectories named `servlets` and `WEB-INF`. We place our HTML files for this servlet chapter in the `servlets` directory. Copy the `WelcomeServlet.html` file into the `servlets` directory. In the `WEB-INF` directory, create the subdirectory `classes`, then copy the `web.xml` file into the `WEB-INF` directory, and copy the `WelcomeServlet.class` file, including all its package name directories, into the `classes` directory. Thus, the directory and file structure under the `webapps` directory should be as shown in Fig. 9.10 (file names are in italics).

**Testing and Debugging Tip 9.3**

Restart the Tomcat server after modifying the `web.xml` deployment descriptor file. Otherwise, Tomcat will not recognize your new Web application.

After the files are placed in the proper directories, start the Tomcat server, open your browser and type the following URL—

```
http://localhost:8080/advjhtpl/servlets/WelcomeServlet.html
```

—to load `WelcomeServlet.html` into the Web browser. Then, click the **Get HTML Document** button to invoke the servlet. You should see the results shown in Fig. 9.6. You can try this servlet from several different Web browsers to demonstrate that the results are the same across Web browsers.

---

**Fig. 9.10** Web application directory and file structure for `WelcomeServlet`.  
```
advjhtpl
  servlets
    WelcomeServlet.html
WEB-INF
  web.xml
  classes
    com
deitel
      advjhtpl
      servlets
        WelcomeServlet.class
```
Common Programming Error 9.2

Not placing servlet or other class files in the appropriate package directory structure prevents the server from locating those classes properly. This, in turn, results in an error response to the client Web browser. This error response normally is “Not Found (404)” in Netscape Navigator and “The page cannot be found” plus an explanation in Microsoft Internet Explorer.

Actually, the HTML file in Fig. 9.6 was not necessary to invoke this servlet. A get request can be sent to a server simply by typing the URL in the Web browser. In fact, that is exactly what you are doing when you request a Web page in the browser. In this example, you can type

http://localhost:8080/advjhttp1/welcome1

in the Address or Location field of your browser to invoke the servlet directly.

Testing and Debugging Tip 9.4

You can test a servlet that handles HTTP get requests by typing the URL that invokes the servlet directly into your browser’s Address or Location field.

9.4 Handling HTTP get Requests Containing Data

When requesting a document or resource from a Web server, it is possible to supply data as part of the request. The servlet WelcomeServlet2 of Fig. 9.11 responds to an HTTP get request that contains a name supplied by the user. The servlet uses the name as part of the response to the client.

```java
// Fig. 9.11: WelcomeServlet2.java
// Processing HTTP get requests containing data.
package com.deitel.advjhttp1.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class WelcomeServlet2 extends HttpServlet {

  // process "get" request from client
  protected void doGet( HttpServletRequest request, 
    HttpServletResponse response ) 
    throws ServletException, IOException 
  { 
    String firstName = request.getParameter( "firstname" );
    response.setContentType( "text/html" );
    PrintWriter out = response.getWriter();
  }

Fig. 9.11 WelcomeServlet2 responds to a get request that contains data (part 1 of 2).
```
Parameters are passed as name/value pairs in a get request. Line 16 demonstrates how to obtain information that was passed to the servlet as part of the client request. The request object’s getParameter method receives the parameter name as an argument and returns the corresponding String value, or null if the parameter is not part of the request. Line 41 uses the result of line 16 as part of the response to the client.

The WelcomeServlet2.html document (Fig. 9.12) provides a form in which the user can input a name in the text input element firstname (line 17) and click the Submit button to invoke WelcomeServlet2. When the user presses the Submit button, the values of the input elements are placed in name/value pairs as part of the request to the server. In the second screen capture of Fig. 9.12, notice that the browser appended

?firstname=Paul

to the end of the action URL. The ? separates the query string (i.e., the data passed as part of the get request) from the rest of the URL in a get request. The name/value pairs are passed with the name and the value separated by =. If there is more than one name/value pair, each name/value pair is separated by &.
Once again, we use our advjhttp1 context root to demonstrate the servlet of Fig. 9.11. Place WelcomeServlet2.html in the servlets directory created in Section 9.3.2. Place WelcomeServlet2.class in the classes subdirectory of WEB-INF in the

---

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- Fig. 9.12: WelcomeServlet2.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Processing get requests with data</title>
</head>
<body>
<form action = "/advjhttp1/welcome2" method = "get">
<p><label>
Type your first name and press the Submit button
<br />
<input type = "text" name = "firstname" />
<input type = "submit" value = "Submit" />
</label></p>
</form>
</body>
</html>
```

Fig. 9.12 HTML document in which the form’s action invokes WelcomeServlet2 through the alias welcome2 specified in web.xml.

form data specified in URL’s query string as part of a get request
advjhttp1 context root. Remember that classes in a package must be placed in the appropriate package directory structure. Then, edit the web.xml deployment descriptor in the WEB-INF directory to include the information specified in Fig. 9.13. This table contains the information for the servlet and servlet-mapping elements that you will add to the web.xml deployment descriptor. You should not type the italic text into the deployment descriptor. Restart Tomcat and type the following URL in your Web browser:

```
http://localhost:8080/advjhttp1/servlets/WelcomeServlet2.html
```

Type your name in the text field of the Web page, then click Submit to invoke the servlet.

Once again, note that the get request could have been typed directly into the browser’s Address or Location field as follows:

```
http://localhost:8080/advjhttp1/welcome2?firstname=Paul
```

Try it with your own name.

### 9.5 Handling HTTP post Requests

An HTTP post request is often used to post data from an HTML form to a server-side form handler that processes the data. For example, when you respond to a Web-based survey, a post request normally supplies the information you specify in the HTML form to the Web server.

Browsers often cache (save on disk) Web pages so they can quickly reload the pages. If there are no changes between the last version stored in the cache and the current version on the Web, this helps speed up your browsing experience. The browser first asks the server if the document has changed or expired since the date the file was cached. If not, the browser loads the document from the cache. Thus, the browser minimizes the amount of data that must be downloaded for you to view a Web page. Browsers typically do not cache the server’s response to a post request, because the next post might not return the same result. For example, in a survey, many users could visit the same Web page and respond to a question. The survey results could then be displayed for the user. Each new response changes the overall results of the survey.

<table>
<thead>
<tr>
<th>Descriptor element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>servlet element</td>
<td></td>
</tr>
<tr>
<td>servlet-name</td>
<td>welcome2</td>
</tr>
<tr>
<td>description</td>
<td>Handling HTTP get requests with data.</td>
</tr>
<tr>
<td>servlet-class</td>
<td>com.deitel.advjhttp1.servlets.WelcomeServlet2</td>
</tr>
<tr>
<td>servlet-mapping element</td>
<td></td>
</tr>
<tr>
<td>servlet-name</td>
<td>welcome2</td>
</tr>
<tr>
<td>url-pattern</td>
<td>/welcome2</td>
</tr>
</tbody>
</table>

*Fig. 9.13* Deployment descriptor information for servlet WelcomeServlet2.
When you use a Web-based search engine, the browser normally supplies the information you specify in an HTML form to the search engine with a get request. The search engine performs the search, then returns the results to you as a Web page. Such pages are often cached by the browser in case you perform the same search again. As with post requests, get requests can supply parameters as part of the request to the Web server.

The WelcomeServlet3 servlet of Fig. 9.14 is identical to the servlet of Fig. 9.11, except that it defines a doPost method (line 12) to respond to post requests rather than a doGet method. The default functionality of doPost is to indicate a “Method not allowed” error. We override this method to provide custom post request processing. Method doPost receives the same two arguments as doGet—an object that implements interface HttpServletRequest to represent the client’s request and an object that implements interface HttpServletResponse to represent the servlet’s response. As with doGet, method doPost throws a ServletException if it is unable to handle a client’s request and throws an IOException if a problem occurs during stream processing.

```java
// Fig. 9.14: WelcomeServlet3.java
// Processing post requests containing data.
package com.deitel.advjhtp1.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class WelcomeServlet3 extends HttpServlet {

    // process "post" request from client
    protected void doPost( HttpServletRequest request,
                           HttpServletResponse response )
                           throws ServletException, IOException {
        String firstName = request.getParameter( "firstname" );
        response.setContentType( "text/html" );
        PrintWriter out = response.getWriter();

        // send XHTML page to client
        // start XHTML document
        out.println( "<?xml version = "1.0"?>" );
        out.println( "<!DOCTYPE html PUBLIC "-//W3C//DTD " +
                      "XHTML 1.0 Strict//EN" "http://www.w3.org" +
                      "/TR/xhtml1/DTD/xhtml1-strict.dtd">" );
        out.println( "<html xmlns = "http://www.w3.org/1999/xhtml">" );

        // head section of document
        out.println( "<head>" );
        // head section of document
        out.println( "<head>" );
```

Fig. 9.14 WelcomeServlet3 responds to a post request that contains data (part 1 of 2).
WelcomeServlet3.html (Fig. 9.15) provides a form (lines 13–21) in which the user can input a name in the text input element firstname (line 17), then click the Submit button to invoke WelcomeServlet3. When the user presses the Submit button, the values of the input elements are sent to the server as part of the request. However, note that the values are not appended to the request URL. Note that the form’s method in this example is post. Also, note that a post request cannot be typed into the browser’s Address or Location field and users cannot bookmark post requests in their browsers.

```java
out.println(
    "<title>Processing post requests with data</title>" );
out.println( "</head>" );

// body section of document
out.println( "<body>" );
out.println( "<h1>Hello " + firstName + ",<br />" );
out.println( "Welcome to Servlets!</h1>" );
out.println( "</body>" );

// end XHTML document
out.println( "</html>" );
out.close(); // close stream to complete the page
```

**WelcomeServlet3** responds to a post request that contains data (part 2 of 2).

Fig. 9.15 HTML document in which the form’s action invokes WelcomeServlet3 through the alias welcome3 specified in web.xml (part 1 of 2).
We use our advjhtp1 context root to demonstrate the servlet of Fig. 9.14. Place WelcomeServlet3.html in the servlets directory created in Section 9.3.2. Place WelcomeServlet3.class in the classes subdirectory of WEB-INF in the advjhtp1 context root. Then, edit the web.xml deployment descriptor in the WEB-INF directory to include the information specified in Fig. 9.16. Restart Tomcat and type the following URL in your Web browser:


Type your name in the text field of the Web page, then click Submit to invoke the servlet.

<table>
<thead>
<tr>
<th>Descriptor element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>servlet element</td>
<td>welcome3</td>
</tr>
<tr>
<td>servlet-name</td>
<td>welcome3</td>
</tr>
<tr>
<td>description</td>
<td>Handling HTTP post requests with data.</td>
</tr>
<tr>
<td>servlet-class</td>
<td>com.deitel.advjhtp1.servlets.WelcomeServlet3</td>
</tr>
<tr>
<td>servlet-mapping element</td>
<td>welcome3</td>
</tr>
<tr>
<td>url-pattern</td>
<td>/welcome3</td>
</tr>
</tbody>
</table>

Fig. 9.15 HTML document in which the form’s action invokes WelcomeServlet3 through the alias welcome3 specified in web.xml (part 2 of 2).

Fig. 9.16 Deployment descriptor information for servlet WelcomeServlet3.
9.6 Redirecting Requests to Other Resources

Sometimes it is useful to redirect a request to a different resource. For example, a servlet could determine the type of the client browser and redirect the request to a Web page that was designed specifically for that browser. The `RedirectServlet` of Fig. 9.17 receives a page parameter as part of a `get` request, then uses that parameter to redirect the request to a different resource.

```java
// Fig. 9.17: RedirectServlet.java
// Redirecting a user to a different Web page.
package com.deitel.advjhtp1.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class RedirectServlet extends HttpServlet {

    // process "get" request from client
    protected void doGet( HttpServletRequest request,
                          HttpServletResponse response )
                        throws ServletException, IOException
    {
        String location = request.getParameter( "page" );

        if ( location != null )
        {
            if ( location.equals( "deitel" ) )
                response.sendRedirect( "http://www.deitel.com" );
            else
                if ( location.equals( "welcome1" ) )
                    response.sendRedirect( "welcome1" );

        // code that executes only if this servlet
        // does not redirect the user to another page
        response.setContentType( "text/html" );
        PrintWriter out = response.getWriter();

        // start XHTML document
        out.println( "<?xml version = "1.0"?>" );
        out.println( "<!DOCTYPE html PUBLIC -//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">" );
        out.println( "<html xmlns = "http://www.w3.org/1999/xhtml">" );

        // head section of document
        out.println( "<head>" );
        out.println( "<title>Invalid page</title>" );
    }
}
```

Fig. 9.17 Redirecting requests to other resources (part 1 of 2).
Line 16 obtains the page parameter from the request. If the value returned is not null, the if/else structure at lines 20–24 determines if the value is either “deitel” or “welcome1.” If the value is “deitel,” the response object’s sendRedirect method (line 21) redirects the request to www.deitel.com. If the value is “welcome1,” line 24 redirect the request to the servlet of Fig. 9.5. Note that line 24 does not explicitly specify the advjhttp1 context root for our Web application. When a servlet uses a relative path to reference another static or dynamic resource, the servlet assumes the same base URL and context root as the one that invoked the servlet—unless a complete URL is specified for the resource. So, line 24 actually is requesting the resource located at

http://localhost:8080/advjhttp1/welcome1

Similarly, line 51 actually is requesting the resource located at

http://localhost:8080/advjhttp1/servlets/RedirectServlet.html

Software Engineering Observation 9.6

Using relative paths to reference resources in the same context root makes your Web application more flexible. For example, you can change the context root without making changes to the static and dynamic resources in the application.

Once method sendRedirect executes, processing of the original request by the RedirectServlet terminates. If method sendRedirect is not called, the remainder of method doPost outputs a Web page indicating that an invalid request was made. The page allows the user to try again by returning to the XHTML document of Fig. 9.18. Note that one of the redirects is sent to a static XHTML Web page and the other is sent to a servlet.

The RedirectServlet.html document (Fig. 9.18) provides two hyperlinks (lines 15–16 and 17–18) that allow the user to invoke the servlet RedirectServlet. Note that each hyperlink specifies a page parameter as part of the URL. To demonstrate passing an invalid page, you can type the URL into your browser with no value for the page parameter.
We use our `advjhttp1` context root to demonstrate the servlet of Fig. 9.17. Place `RedirectServlet.html` in the `servlets` directory created in Section 9.3.2. Place `RedirectServlet.class` in the `classes` subdirectory of `WEB-INF` in the `advjhttp1` context root. Then, edit the `web.xml` deployment descriptor in the `WEB-INF`
directory to include the information specified in Fig. 9.19. Restart Tomcat, and type the following URL in your Web browser:

http://localhost:8080/advjhtp1/servlets/RedirectServlet.html

Click a hyperlink in the Web page to invoke the servlet.

When redirecting requests, the request parameters from the original request are passed as parameters to the new request. Additional request parameters also can be passed. For example, the URL passed to `sendRedirect` could contain name/value pairs. Any new parameters are added to the existing parameters. If a new parameter has the same name as an existing parameter, the new parameter value takes precedence over the original value. However, all the values are still passed. In this case, the complete set of values for a given parameter name can be obtained by calling method `getParameterValues` from interface `HttpServletRequest`. This method receives the parameter name as an argument and returns an array of `Strings` containing the parameter values in order from most recent to least recent.

### 9.7 Session Tracking

Many e-businesses can personalize users’ browsing experiences, tailoring Web pages to their users’ individual preferences and letting users bypass irrelevant content. This is done by tracking the consumer’s movement through the Internet and combining that data with information provided by the consumer, which could include billing information, interests and hobbies, among other things. **Personalization** is making it easier and more pleasant for many people to surf the Internet and find what they want. Consumers and companies can benefit from the unique treatment resulting from personalization. Providing content of special interest to your visitor can help establish a relationship that you can build upon each time that person returns to your site. Targeting consumers with personal offers, advertisements, promotions and services may lead to more customer loyalty—many customers enjoy the individual attention that a customized site provides. Originally, the Internet lacked personal assistance when compared with the individual service often experienced in bricks-and-mortar stores. Sophisticated technology helps many Web sites offer a personal touch

<table>
<thead>
<tr>
<th>Descriptor element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>servlet</code> element</td>
<td></td>
</tr>
<tr>
<td><code>servlet-name</code></td>
<td>redirect</td>
</tr>
<tr>
<td><code>description</code></td>
<td>Redirecting to static Web pages and other servlets.</td>
</tr>
<tr>
<td><code>servlet-class</code></td>
<td>com.deitel.advjhtp1.servlets.RedirectServlet</td>
</tr>
<tr>
<td><code>servlet-mapping</code> element</td>
<td></td>
</tr>
<tr>
<td><code>servlet-name</code></td>
<td>redirect</td>
</tr>
<tr>
<td><code>url-pattern</code></td>
<td>/redirect</td>
</tr>
</tbody>
</table>

*Fig. 9.19  Deployment descriptor information for servlet RedirectServlet.*
to their visitors. For example, Web sites such as MSN.com and CNN.com allow you to customize their home page to suit your needs. Online shopping sites often customize their Web pages to individuals, and such sites must distinguish between clients so the company can determine the proper items and charge the proper amount for each client. Personalization is important for Internet marketing and for managing customer relationships to increase customer loyalty.

Hand in hand with the promise of personalization, however, comes the problem of privacy invasion. What if the e-business to which you give your personal data sells or gives those data to another organization without your knowledge? What if you do not want your movements on the Internet to be tracked by unknown parties? What if an unauthorized party gains access to your private data, such as credit-card numbers or medical history? These are some of the many questions that must be addressed by consumers, e-businesses and lawmakers alike.

As we have discussed, the request/response mechanism of the Web is based on HTTP. Unfortunately, HTTP is a stateless protocol—it does not support persistent information that could help a Web server determine that a request is from a particular client. As far as a Web server is concerned, every request could be from the same client or every request could be from a different client. Thus, sites like MSN.com and CNN.com need a mechanism to identify individual clients. To help the server distinguish between clients, each client must identify itself to the server. There are a number of popular techniques for distinguishing between clients. We introduce two techniques to track clients individually—cookies (Section 9.7.1) and session tracking (Section 9.7.2). Two other techniques not discussed in this chapter are using input form elements of type "hidden" and URL rewriting. With "hidden" form elements, the servlet can write session-tracking data into a form in the Web page it returns to the client to satisfy a prior request. When the user submits the form in the new Web page, all the form data, including the "hidden" fields, are sent to the form handler on the server. With URL rewriting, the servlet embeds session-tracking information as get parameters directly in the URLs of hyperlinks that the user might click to make the next request to the Web server.

9.7.1 Cookies

A popular way to customize Web pages is via cookies. Browsers can store cookies on the user’s computer for retrieval later in the same browsing session or in future browsing sessions. For example, cookies could be used in a shopping application to store unique identifiers for the users. When users add items to their online shopping carts or perform other tasks resulting in a request to the Web server, the server receives cookies containing unique identifiers for each user. The server then uses the unique identifier to locate the shopping carts and perform the necessary processing. Cookies could also be used to indicate the client’s shopping preferences. When the servlet receives the client’s next communication, the servlet can examine the cookie(s) it sent to the client in a previous communication, identify the client’s preferences and immediately display products of interest to the client.

Cookies are text-based data that are sent by servlets (or other similar server-side technologies) as part of responses to clients. Every HTTP-based interaction between a client and a server includes a header containing information about the request (when the communication is from the client to the server) or information about the response (when the communication is from the server to the client). When an HttpServlet receives a request,
the header includes information such as the request type (e.g., `get` or `post`) and the cookies that are sent by the server to be stored on the client machine. When the server formulates its response, the header information includes any cookies the server wants to store on the client computer and other information such as the MIME type of the response.

**Testing and Debugging Tip 9.5**

Some clients do not accept cookies. When a client declines a cookie, the Web site or the browser application can inform the client that the site may not function correctly without cookies enabled.

Depending on the maximum age of a cookie, the Web browser either maintains the cookie for the duration of the browsing session (i.e., until the user closes the Web browser) or stores the cookie on the client computer for future use. When the browser requests a resource from a server, cookies previously sent to the client by that server are returned to the server as part of the request formulated by the browser. Cookies are deleted automatically when they expire (i.e., reach their maximum age).

Figure 9.20 demonstrates cookies. The example allows the user to select a favorite programming language and `post` the choice to the server. The response is a Web page in which the user can select another favorite language or click a link to view a list of book recommendations. When the user selects the list of book recommendations, a `get` request is sent to the server. The cookies previously stored on the client are read by the servlet and used to form a Web page containing the book recommendations.

**CookieServlet** (Fig. 9.20) handles both the `get` and the `post` requests. The **CookieSelectLanguage.html** document of Fig. 9.21 contains four radio buttons (C, C++, Java and VB 6) and a **Submit** button. When the user presses Submit, the **CookieServlet** is invoked with a `post` request. The servlet adds a cookie containing the selected language to the response header and sends an XHTML document to the client. Each time the user clicks **Submit**, a cookie is sent to the client.

Line 11 defines `Mapbooks` as a `HashMap` (package `java.util`) in which we store key/value pairs that use the programming language as the key and the ISBN number of the recommended book as the value. The **CookieServlet init** method (line 14–20) populates books with four key/value pairs of books. Method **doPost** (lines 24–69) is invoked in response to the `post` request from the XHTML document of Fig. 9.21. Line 28 uses method **getParameter** to obtain the user’s `language` selection (the value of the selected radio button on the Web page). Line 29 obtains the ISBN number for the selected language from `books`.

```java
// Fig. 9.20: CookieServlet.java
// Using cookies to store data on the client computer.
package com.deitel.advjhtp1.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

// Fig. 9.20: Storing user data on the client computer with cookies (part 1 of 4).
```
562

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

Servlets

Chapter 9

public class CookieServlet extends HttpServlet {
private final Map books = new HashMap();

Fig. 9.20

// initialize Map books
public void init()
{
books.put( "C", "0130895725" );
books.put( "C++", "0130895717" );
books.put( "Java", "0130125075" );
books.put( "VB6", "0134569555" );
}
// receive language selection and send cookie containing
// recommended book to the client
protected void doPost( HttpServletRequest request,
HttpServletResponse response )
throws ServletException, IOException
{
String language = request.getParameter( "language" );
String isbn = books.get( language ).toString();
Cookie cookie = new Cookie( language, isbn );
response.addCookie( cookie ); // must precede getWriter
response.setContentType( "text/html" );
PrintWriter out = response.getWriter();
// send XHTML page to client
// start XHTML document
out.println( "<?xml version = \"1.0\"?>" );
out.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
"XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
"/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
out.println(
"<html xmlns = \"http://www.w3.org/1999/xhtml\">" );
// head section of document
out.println( "<head>" );
out.println( "<title>Welcome to Cookies</title>" );
out.println( "</head>" );
// body section of document
out.println( "<body>" );
out.println( "<p>Welcome to Cookies! You selected " +
language + "</p>" );
out.println( "<p><a href = " +
"\"/advjhtp1/servlets/CookieSelectLanguage.html\">" +
"Click here to choose another language</a></p>" );

Storing user data on the client computer with cookies (part 2 of 4).


Fig. 9.20 Storing user data on the client computer with cookies (part 3 of 4).
Line 30 creates a new `Cookie` object (package `javax.servlet.http`), using the `language` and `isbn` values as the cookie name and cookie value, respectively. The cookie name identifies the cookie; the cookie value is the information associated with the cookie. Browsers that support cookies must be able to store a minimum of 20 cookies per Web site and 300 cookies per user. Browsers may limit the cookie size to 4K (4096 bytes). Each cookie stored on the client includes a domain. The browser sends a cookie only to the domain stored in the cookie.

Software Engineering Observation 9.7
Browser users can disable cookies, so Web applications that use cookies may not function properly for clients with cookies disabled.

Software Engineering Observation 9.8
By default, cookies exist only for the current browsing session (until the user closes the browser). To make cookies persist beyond the current session, call `Cookie` method `setMaxAge` to indicate the number of seconds until the cookie expires.

Line 32 adds the cookie to the `response` with method `addCookie` of interface `HttpServletResponse`. Cookies are sent to the client as part of the HTTP header. The header information is always provided to the client first, so the cookies should be added to the `response` with `addCookie` before any data is written as part of the response. After the cookie is added, the servlet sends an XHTML document to the client (see the second screen capture of Fig. 9.21).

Common Programming Error 9.3
Writing response data to the client before calling method `addCookie` to add a cookie to the response is a logic error. Cookies must be added to the header first.

The XHTML document sent to the client in response to a `post` request includes a hyperlink that invokes method `doGet` (lines 73–123). The method reads any `Cookies` that were written to the client in `doPost`. For each `Cookie` written, the servlet recommends a Deitel book on the subject. Up to four books are displayed on the Web page created by the servlet.

Line 77 retrieves the cookies from the client using `HttpServletRequest` method `getCookies`, which returns an array of `Cookie` objects. When a `get` or `post` operation is performed to invoke a servlet, the cookies associated with that server’s domain are automatically sent to the servlet.
If method `getCookies` does not return `null` (i.e., there were no cookies), lines 106–109 retrieve the name of each `Cookie` using `Cookie` method `getName`, retrieve the value of each `Cookie` using `Cookie` method `getValue` and write a line to the client indicating the name of a recommended book and its ISBN number.

**Software Engineering Observation 9.9**

Normally, each servlet class handles one request type (e.g., `get` or `post`, but not both).

Figure 9.21 shows the XHTML document the user loads to select a language. When the user presses `Submit`, the value of the currently selected radio button is sent to the server as part of the `post` request to the `CookieServlet`, which we refer to as `cookies` in this example.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.21: CookieSelectLanguage.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Using Cookies</title>
</head>
<body>
<form action = "/advjhtp1/cookies" method = "post">
<p>Select a programming language:</p>
<input type = "radio" name = "language" value = "C" /> C  
<input type = "radio" name = "language" value = "C++" /> C++  
<!-- this radio button checked by default -->  
<input type = "radio" name = "language" value = "Java" checked = "checked" /> Java  
<input type = "radio" name = "language" value = "VB6" /> VB 6  
</p>
<p><input type = "submit" value = "Submit" /></p>
</form>
</body>
</html>
```

Fig. 9.21 `CookieSelectLanguage.html` document for selecting a programming language and posting the data to the `CookieServlet` (part 1 of 3).
Fig. 9.21  **CookieSelectLanguage.html** document for selecting a programming language and posting the data to the **CookieServlet** (part 2 of 3).
We use our advjhttp1 context root to demonstrate the servlet of Fig. 9.20. Place CookieSelectLanguage.html in the servlets directory created previously. Place CookieServlet.class in the classes subdirectory of WEB-INF in the advjhttp1 context root. Then, edit the web.xml deployment descriptor in the WEB-INF directory to include the information specified in Fig. 9.22. Restart Tomcat and type the following URL in your Web browser:

http://localhost:8080/advjhttp1/servlets/CookieSelectLanguage.html

Select a language, and press the Submit button in the Web page to invoke the servlet.

Various Cookie methods are provided to manipulate the members of a Cookie. Some of these methods are listed in Fig. 9.23.

Fig. 9.21  CookieSelectLanguage.html document for selecting a programming language and posting the data to the CookieServlet (part 3 of 3).
**Descriptor element** | **Value**
--- | ---
*servlet* element |  
*servlet-name* | cookies  
*description* | Using cookies to maintain state information.  
*servlet-class* | com.deitel.advjhtp1.servlets.CookieServlet

**servlet-mapping** element |  
*servlet-name* | cookies  
*url-pattern* | /cookies

**Fig. 9.22** Deployment descriptor information for servlet *CookieServlet*.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getComment()</code></td>
<td>Returns a <em>String</em> describing the purpose of the cookie (<em>null</em> if no comment has been set with <code>setComment</code>).</td>
</tr>
<tr>
<td><code>getDomain()</code></td>
<td>Returns a <em>String</em> containing the cookie’s domain. This determines which servers can receive the cookie. By default, cookies are sent to the server that originally sent the cookie to the client.</td>
</tr>
<tr>
<td><code>getMaxAge()</code></td>
<td>Returns an <em>int</em> representing the maximum age of the cookie in seconds.</td>
</tr>
<tr>
<td><code>getName()</code></td>
<td>Returns a <em>String</em> containing the name of the cookie as set by the constructor.</td>
</tr>
<tr>
<td><code>getPath()</code></td>
<td>Returns a <em>String</em> containing the URL prefix for the cookie. Cookies can be “targeted” to specific URLs that include directories on the Web server. By default, a cookie is returned to services operating in the same directory as the service that sent the cookie or a subdirectory of that directory.</td>
</tr>
<tr>
<td><code>getSecure()</code></td>
<td>Returns a <em>boolean</em> value indicating if the cookie should be transmitted using a secure protocol (<em>true</em>).</td>
</tr>
<tr>
<td><code>getValue()</code></td>
<td>Returns a <em>String</em> containing the value of the cookie as set with <code>setValue</code> or the constructor.</td>
</tr>
<tr>
<td><code>getVersion()</code></td>
<td>Returns an <em>int</em> containing the version of the cookie protocol used to create the cookie. A value of 0 (the default) indicates the original cookie protocol as defined by Netscape. A value of 1 indicates the current version, which is based on Request for Comments (RFC) 2109.</td>
</tr>
<tr>
<td><code>setComment(String)</code></td>
<td>The comment describing the purpose of the cookie that is presented by the browser to the user. (Some browsers allow the user to accept cookies on a per-cookie basis.)</td>
</tr>
</tbody>
</table>

**Fig. 9.23** Important methods of class *Cookie* (part 1 of 2).
Chapter 9  Servlets  569

9.7.2 Session Tracking with HttpSession

Java provides enhanced session tracking support with the servlet API’s HttpSession interface. To demonstrate basic session-tracking techniques, we modified the servlet from Fig. 9.20 to use HttpSession objects (Fig. 9.24). Once again, the servlet handles both get and post requests. The document SessionSelectLanguage.html of Fig. 9.25 contains four radio buttons (C, C++, Java and VB 6) and a Submit button. When the user presses Submit, SessionServlet is invoked with a post request. The servlet responds by creating an object of type HttpSession for the client (or using an existing session for the client) and adds the selected language and an ISBN number for the recommended book to the HttpSession object. Then, the servlet sends an XHTML page to the client. Each time the user clicks Submit, a new language/ISBN pair is added to the HttpSession object.

Software Engineering Observation 9.10

A servlet should not use instance variables to maintain client state information, because clients accessing that servlet in parallel might overwrite the shared instance variables. Servlets should maintain client state information in HttpSession objects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setDomain(String)</td>
<td>This determines which servers can receive the cookie. By default, cookies are sent to the server that originally sent the cookie to the client. The domain is specified in the form &quot;.deitel.com&quot;, indicating that all servers ending with .deitel.com can receive this cookie.</td>
</tr>
<tr>
<td>setMaxAge(int)</td>
<td>Sets the maximum age of the cookie in seconds.</td>
</tr>
<tr>
<td>setPath(String)</td>
<td>Sets the “target” URL prefix indicating the directories on the server that lead to the services that can receive this cookie.</td>
</tr>
<tr>
<td>setSecure(boolean)</td>
<td>A true value indicates that the cookie should only be sent using a secure protocol.</td>
</tr>
<tr>
<td>setValue(String)</td>
<td>Sets the value of a cookie.</td>
</tr>
<tr>
<td>setVersion(int)</td>
<td>Sets the cookie protocol for this cookie.</td>
</tr>
</tbody>
</table>

### Fig. 9.23
Important methods of class Cookie (part 2 of 2).

### Fig. 9.24
Maintaining state information with HttpSession objects (part 1 of 4).

```java
// Fig. 9.24: SessionServlet.java
// Using HttpSession to maintain client state information.
package com.deitel.advjhtp1.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
```

### Fig. 9.25
SessionSelectLanguage.html contains four radio buttons (C, C++, Java and VB 6) and a Submit button.
**public class SessionServlet extends HttpServlet {**

private final Map books = new HashMap();

// initialize Map books
public void init()
{
    books.put( "C", "0130895725" );
    books.put( "C++", "0130895717" );
    books.put( "Java", "0130125075" );
    books.put( "VB6", "0134569555" );
}

// receive language selection and create HttpSession object
// containing recommended book for the client
protected void doPost( HttpServletRequest request,
HttpServletResponse response )
throws ServletException, IOException
{
    String language = request.getParameter( "language" );

    // Get the user's session object.
    // Create a session (true) if one does not exist.
    HttpSession session = request.getSession( true );

    // add a value for user's choice to session
    session.setAttribute( language, books.get( language ) );

    response.setContentType( "text/html" );
    PrintWriter out = response.getWriter();

    // send XHTML page to client
    // start XHTML document
    out.println( "<?xml version = "1.0"?>" );
    out.println( "<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">" );
    out.println( "<html xmlns = "http://www.w3.org/1999/xhtml">" );

    // head section of document
    out.println( "<head>" );
    out.println( "<title>Welcome to Sessions</title>" );
    out.println( "</head>" );

    // body section of document
    out.println( "<body>" );
    out.println( "<p>Welcome to Sessions! You selected " + language + ".</p>" );
// display information about the session
out.println( "<p>Your unique session ID is: " +
    session.getId() + "<br />
);  

out.println("This " + ( session.isNew() ? "is" : "is not" ) +
    " a new session<br />
);  

out.println( "The session was created at: " +
    new Date( session.getCreationTime() ) + "<br />
);  

out.println( "You last accessed the session at: " +
    new Date( session.getLastAccessedTime() ) + "<br />
);  

out.println( "The maximum inactive interval is: " +
    session.getMaxInactiveInterval() + " seconds</p>" );  

out.println("<p><a href="servlets/SessionSelectLanguage.html">Click here to choose another language</a></p>" );  

out.println("<p><a href="sessions">Click here to get book recommendations</a></p>" );  

out.println( "</body>" );  

// end XHTML document
out.println("</html>" );
out.close();  // close stream
}

// read session attributes and create XHTML document
// containing recommended books
protected void doGet( HttpServletRequest request,
    HttpServletResponse response )
    throws ServletException, IOException
{
    // Get the user's session object.
    // Do not create a session (false) if one does not exist.
    HttpSession session = request.getSession( false );

    // get names of session object's values
    Enumeration valueNames;

    if ( session != null )
        valueNames = session.getAttributeNames();
    else
        valueNames = null;

    PrintWriter out = response.getWriter();
    response.setContentType( "text/html" );

    // start XHTML document
    out.println( "<?xml version = "1.0"?>" );
Most of class `SessionServlet` is identical to `CookieServlet` (Fig. 9.20), so we concentrate on only the new features here. When the user selects a language from the document `SessionSelectLanguage.html` (Fig. 9.25) and presses `Submit`, method `doPost` (lines 24–90) is invoked. Line 28 gets the user’s `language` selection. Then, line 32 uses method `getSession` of interface `HttpServletRequest` to obtain the
**HttpSession** object for the client. If the server has an existing **HttpSession** object for the client from a previous request, method `getSession` returns that **HttpSession** object. Otherwise, the `true` argument to method `getSession` indicates that the servlet should create a unique new **HttpSession** object for the client. A `false` argument would cause method `getSession` to return `null` if the **HttpSession** object for the client did not already exist. Using a `false` argument could help determine whether a client has logged into a Web application.

Like a cookie, an **HttpSession** object can store name/value pairs. In session terminology, these are called **attributes**, and they are placed into an **HttpSession** object with method `setAttribute`. Line 35 uses `setAttribute` to put the language and the corresponding recommended book’s ISBN number into the **HttpSession** object. One of the primary benefits of using **HttpSession** objects rather than cookies is that **HttpSession** objects can store any object (not just **Strings**) as the value of an attribute. This allows Java programmers flexibility in determining the type of state information they wish to maintain for clients of their Web applications. If an attribute with a particular name already exists when `setAttribute` is called, the object associated with that attribute name is replaced.

Software Engineering Observation 9.11

Name/value pairs added to an **HttpSession** object with `setAttribute` remain available until the client’s current browsing session ends or until the session is invalidated explicitly by a call to the **HttpSession** object’s `invalidate` method. Also, if the servlet container is restarted, these attributes may be lost.

After the values are added to the **HttpSession** object, the servlet sends an XHTML document to the client (see the second screen capture of Fig. 9.25). In this example, the document contains various information about the **HttpSession** object for the current client. Line 64 uses **HttpSession** method `getID` to obtain the session’s unique ID number. Line 67 determines whether the session is new or already exists with method `isNew`, which returns `true` or `false`. Line 71 obtains the time at which the session was created with method `getCreationTime`. Line 74 obtains the time at which the session was last accessed with method `getLast AccessedTime`. Line 77 uses method `getMaxInactiveInterval` to obtain the maximum amount of time that an **HttpSession** object can be inactive before the servlet container discards it.

The XHTML document sent to the client in response to a `post` request includes a hyperlink that invokes method `doGet` (lines 94–159). The method obtains the **HttpSession** object for the client with method `getSession` (line 100). We do not want to make any recommendations if the client does not have an existing **HttpSession** object. So, this call to `getSession` uses a `false` argument. Thus, `getSession` returns an **HttpSession** object only if one already exists for the client.

If method `getSession` does not return `null`, line 106 uses **HttpSession** method `getAttributeNames` to retrieve an **Enumeration** of the attribute names (i.e., the names used as the first argument to **HttpSession** method `setAttribute`). Each name is passed as an argument to **HttpSession** method `getAttribute` (line 141) to retrieve the ISBN of a book from the **HttpSession** object. Method `getAttribute` receives the name and returns an **Object** reference to the corresponding value. Next, a line is written in the response to the client containing the title and ISBN number of the recommended book.

Figure 9.25 shows the XHTML document the user loads to select a language. When the user presses **Submit**, the value of the currently selected radio button is sent to the
server as part of the post request to the SessionServlet, which we refer to as sessions in this example.

We use our advjhtpl context root to demonstrate the servlet of Fig. 9.24. Place SessionSelectLanguage.html in the servlets directory created previously. Place SessionServlet.class in the classes subdirectory of WEB-INF in the advjhtpl context root. Then, edit the web.xml deployment descriptor in the WEB-INF directory to include the information specified in Fig. 9.26. Restart Tomcat and type the following URL in your Web browser:

```
http://localhost:8080/advjhtpl/servlets/
SessionSelectLanguage.html
```

Select a language, and press the Submit button in the Web page to invoke the servlet.

---

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.25: SessionSelectLanguage.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Using Sessions</title>
</head>
<body>
<form action = "/advjhtpl/sessions" method = "post">
<p>Select a programming language:</p>
<input type = "radio" name = "language"
value = "C" />C <br />
<input type = "radio" name = "language"
value = "C++" />C++ <br />
<!-- this radio button checked by default -->
<input type = "radio" name = "language"
value = "Java" checked = "checked" />Java<br />
<input type = "radio" name = "language"
value = "VB6" />VB 6

<p><input type = "submit" value = "Submit" /></p>
</form>
</body>
</html>
```

Fig. 9.25 SessionSelectLanguage.html document for selecting a programming language and posting the data to the SessionServlet (part 1 of 4).
Fig. 9.25  SessionSelectLanguage.html document for selecting a programming language and posting the data to the SessionServlet (part 2 of 4).
Fig. 9.25  *SessionSelectLanguage.html* document for selecting a programming language and posting the data to the *SessionServlet* (part 3 of 4).
9.8 Multi-tier Applications: Using JDBC from a Servlet

Servlets can communicate with databases via JDBC (Java Database Connectivity). As we discussed in Chapter 8, JDBC provides a uniform way for a Java program to connect with a variety of databases in a general manner without having to deal with the specifics of those database systems.

Many of today’s applications are three-tier distributed applications, consisting of a user interface, business logic and database access. The user interface in such an application is often created using HTML, XHTML (as shown in this chapter) or Dynamic HTML. In some cases, Java applets are also used for this tier. HTML and XHTML are the preferred mechanisms for representing the user interface in systems where portability is a concern. Because HTML is supported by all browsers, designing the user interface to be accessed through a Web browser guarantees portability across all platforms that have browsers. Using the networking provided automatically by the browser, the user interface can communicate with the middle-tier business logic. The middle tier can then access the database...
to manipulate the data. The three tiers can reside on separate computers that are connected to a network.

In multi-tier architectures, Web servers often represent the middle tier. They provide the business logic that manipulates data from databases and that communicates with client Web browsers. Servlets, through JDBC, can interact with popular database systems. Developers do not need to be familiar with the specifics of each database system. Rather, developers use SQL-based queries and the JDBC driver handles the specifics of interacting with each database system.

The SurveyServlet of Fig. 9.27 and the Survey.html document of Fig. 9.28 demonstrate a three-tier distributed application that displays the user interface in a browser using XHTML. The middle tier is a Java servlet that handles requests from the client browser and provides access to the third tier—a Cloudscape database accessed via JDBC. The servlet in this example is a survey servlet that allows users to vote for their favorite animal. When the servlet receives a post request from the Survey.html document, the servlet updates the total number of votes for that animal in the database and returns a dynamically generated XHTML document containing the survey results to the client.

```java
// Fig. 9.27: SurveyServlet.java
// A Web-based survey that uses JDBC from a servlet.
package com.deitel.advjhtp1.servlets;

import java.io.*;
import java.text.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SurveyServlet extends HttpServlet {
    private Connection connection;
    private PreparedStatement updateVotes, totalVotes, results;

    // set up database connection and prepare SQL statements
    public void init( ServletConfig config )
        throws ServletException
    {
        try {
            Class.forName( "COM.cloudscape.core.RmiJdbcDriver" );
            connection = DriverManager.getConnection(
                "jdbc:rmi:jdbc:cloudscape:animalsurvey" );

            // PreparedStatement to add one to vote total for a
            // specific animal
            updateVotes =
                connection.prepareStatement(
                    "UPDATE surveyresults SET votes = votes + 1 " +
                    "WHERE id = ?" );
        }
```
// PreparedStatement to sum the votes
    totalVotes =
        connection.prepareStatement(
            "SELECT sum( votes ) FROM surveyresults"
        );

    // PreparedStatement to obtain surveyoption table's data
    results =
        connection.prepareStatement(
            "SELECT surveyoption, votes, id " +
            "FROM surveyresults ORDER BY id"
        );

}  // end of init method

// process survey response
protected void doPost( HttpServletRequest request,
    HttpServletResponse response )
    throws ServletException, IOException
{
    // set up response to client
    response.setContentType( "text/html" );
    PrintWriter out = response.getWriter();
    DecimalFormat twoDigits = new DecimalFormat( "0.00" );

    // start XHTML document
    out.println( "<?xml version = "1.0"?>" );

    out.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" \"http://www.w3.org +
        "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );

    out.println( "<html xmlns = "http://www.w3.org/1999/xhtml\">" );

    // head section of document
    out.println( "<head>" );

    // read current survey response
    int value =
        Integer.parseInt( request.getParameter( "animal" ) );

    // attempt to process a vote and display current results
    try {
// update total for current survey response
updateVotes.setInt(1, value);
updateVotes.executeUpdate();

// get total of all survey responses
ResultSet totalRS = totalVotes.executeQuery();
totalRS.next();
int total = totalRS.getInt(1);

// get results
ResultSet resultsRS = results.executeQuery();
out.println("<title>Thank you!</title> ");
out.println("</head> ");

out.println("<body> ");
out.println("<p>Thank you for participating." );
out.println("<br />Results:</p><pre>

// process results
int votes;

while (resultsRS.next()) {
    out.print(resultsRS.getString(1));
    out.print(":");
    votes = resultsRS.getInt(2);
    out.print(twoDigits.format((double) votes / total * 100));
    out.print("% responses: ");
    out.println(votes);
}
resultsRS.close();
out.print("Total responses: ");
out.print(total);

// end XHTML document
out.println("</pre></body></html> ");
out.close();

// if database exception occurs, return error page
catch (SQLException sqlException) {
    sqlException.printStackTrace();
    out.println("<title>Error</title> ");
    out.println("</head> ");
    out.println("<body><p>Database error occurred. ");
    out.println("Try again later.</p></body></html> ");
    out.close();
}

} // end of doPost method

Fig. 9.27 Multi-tier Web-based survey using XHTML, servlets and JDBC (part 3 of 4).
Lines 12 and 13 begin by declaring a `Connection` reference to manage the database connection and three `PreparedStatement` references for updating the vote count for an animal, totalling all the votes and obtaining the complete survey results.

Servlets are initialized by overriding method `init` (lines 16–54). Method `init` is called exactly once in a servlet’s lifetime, before any client requests are accepted. Method `init` takes a `ServletConfig` argument and throws a `ServletException`. The argument provides the servlet with information about its initialization parameters (i.e., parameters not associated with a request, but passed to the servlet for initializing servlet variables). These parameters are specified in the `web.xml` deployment descriptor file as part of a `servlet` element. Each parameter appears in an `init-param` element of the following form:

```
<init-param>
  <param-name>parameter name goes here</param-name>
  <param-value>parameter value goes here</param-value>
</init-param>
```

Servlets can obtain initialization parameter values by invoking `ServletConfig` method `getInitParameter`, which receives a string representing the name of the parameter.

In this example, the servlet’s `init` method (lines 16–54) performs the connection to the Cloudscape database. Line 21 loads the driver (`COM.cloudscape.core.RmiJdbcDriver`). Lines 22–23 attempt to open a connection to the `animalsurvey` database. The database contains one table (`surveyresults`) that consists of three fields—a unique integer to identify each record called `id`, a string representing the survey option called `surveyoption` and an integer representing the number of votes for a survey option called `votes`. [Note: The examples folder for this chapter contains an SQL script (`animalsurvey.sql`) that you can use to create the `animalsurvey` database for this example. For information on starting the Cloudscape server and executing the SQL script, please refer back to Chapter 8.]

Lines 27–44 create `PreparedStatement` objects called `updateVotes`, `totalVotes` and `results`. The `updateVotes` statement adds one to the `votes`
value for the record with the specified ID. The `totalVotes` statement uses SQL’s built-in `sum` capability to total all the `votes` in the `surveyresults` table. The results statement returns all the data in the `surveyresults` table.

When a user submits a survey response, method `doPost` (lines 57–137) handles the request. Lines 80–81 obtain the survey response, then the `try` block (lines 84–125) attempts to process the response. Lines 87–88 set the first parameter of `PreparedStatement updateVotes` to the survey response and update the database. Lines 91–93 execute `PreparedStatement totalVotes` to retrieve the total number of votes received. Then, lines 96–123 execute `PreparedStatement results` and process the `ResultSet` to create the survey summary for the client. When the servlet container terminates the servlet, method `destroy` (lines 140–154) closes each `PreparedStatement`, then closes the database connection. Figure 9.28 shows survey.html, which invokes `SurveyServlet` with the alias `animalsurvey` when the user submits the form.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 9.28: Survey.html -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Survey</title>
</head>
<body>
<form method = "post" action = "/advjhtp1/animalsurvey">
<p>What is your favorite pet?</p>

<p>
<input type = "radio" name = "animal"
value = "1" />Dog<br />
<input type = "radio" name = "animal"
value = "2" />Cat<br />
<input type = "radio" name = "animal"
value = "3" />Bird<br />
<input type = "radio" name = "animal"
value = "4" />Snake<br />
<input type = "radio" name = "animal"
value = "5" checked = "checked" />None
</p>
<p><input type = "submit" value = "Submit" /></p>
</form>
</body>
</html>
```

**Fig. 9.28** *Survey.html* document that allows users to submit survey responses to *SurveyServlet* (part 1 of 2).
We use our advjhttp1 context root to demonstrate the servlet of Fig. 9.27. Place Survey.html in the servlets directory created previously. Place SurveyServlet.class in the classes subdirectory of WEB-INF in the advjhttp1 context root. Then, edit the web.xml deployment descriptor in the WEB-INF directory to include the information specified in Fig. 9.29. Also, this program cannot execute in Tomcat unless the Web application is aware of the JAR files cloudscape.jar and RmiJdbc.jar that contain the Cloudscape database driver and its supporting classes. The cloudscape.jar file is located in your Cloudscape installation’s lib directory. The RmiJdbc.jar file is located in your Cloudscape installation’s frameworks\RmiJdbc\classes directory. Place copies of these files in the WEB-INF subdirectory lib to make them available to the Web application. Please refer back to Chapter 8 for more information on the set up and configuration of Cloudscape.
A copy of these files should be placed in the `advjhtp1` context root’s `WEB-INF` subdirectory called `lib`. After copying these files, restart Tomcat and type the following URL in your Web browser:

```
http://localhost:8080/advjhtp1/servlets/Survey.html
```

Select a survey response, and press the **Submit** button in the Web page to invoke the servlet.

### 9.9 `HttpUtils` Class

Class **`HttpUtils`** provides three **static** utility methods to simplify servlet programming. These methods are discussed in Fig. 9.30.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getRequestURL</code></td>
<td>This method takes the <code>HttpServletRequest</code> object as an argument and returns a <code>StringBuffer</code> containing the original URL that initiated the request.</td>
</tr>
<tr>
<td><code>parsePostData</code></td>
<td>This method receives an integer and <code>ServletInputStream</code> as arguments. The integer represents the number of bytes in the <code>ServletInputStream</code>. The <code>ServletInputStream</code> contains the key/value pairs posted to the servlet from a form. The method returns a <code>Hashtable</code> containing the key/value pairs.</td>
</tr>
<tr>
<td><code>parseQueryString</code></td>
<td>This method receives a <code>String</code> representing the query string in a <code>get</code> request as an argument and returns a <code>Hashtable</code> containing the key/value pairs in the query string. The value of each key is an array of <code>Strings</code>. The query string can be obtained with <code>HttpServletRequest</code> method <code>getQueryString</code>.</td>
</tr>
</tbody>
</table>

---

**Fig. 9.30  `HttpUtils` class methods.**
9.10 Internet and World Wide Web Resources

This section lists a variety of servlet resources available on the Internet and provides a brief description of each.

java.sun.com/products/servlet/index.html
The servlet page at the Sun Microsystems, Inc., Java Web site provides access to the latest servlet information and servlet resources.

jakarta.apache.org
This is the Apache Project’s home page for the Jakarta Project. Tomcat—the servlets and JavaServer Pages reference implementation—is one of many subprojects of the Jakarta Project.

jakarta.apache.org/tomcat/index.html
Home page for the Tomcat servlets and JavaServer Pages reference implementation.

java.apache.org
This is the Apache Project’s home page for all Java-related technologies. This site provides access to many Java packages useful to servlet and JSP developers.

www.servlets.com
This is the Web site for the book Java Servlet Programming published by O’Reilly. The book provides a variety of resources. This book is an excellent resource for programmers who are learning servlets.

theserverside.com
TheServerSide.com is dedicated to information and resources for Enterprise Java.

www.servletsource.com
ServletSource.com is a general servlet resource site containing code, tips, tutorials and links to many other Web sites with information on servlets.

www.cookiecentral.com
A good all-around resource site for cookies.

developer.netscape.com/docs/manuals/communicator/jsguide4/cookies.htm
A description of Netscape cookies.

www.javacorporate.com
Home of the open-source Expresso Framework, which includes a library of extensible servlet components to help speed Web application development.

www.servlet.com/srvdev.jhtml
ServletInc’s Servlet Developers Forum provides resources for server-side Java developers and information about Web servers that support servlet technologies.

www.servletforum.com
ServletForum.com is a newsgroup where you can post questions and have them answered by your peers.

www.coolservlets.com
Provides free open-source Java servlets.

www.cetus-links.org/oo_java_servlets.html
Provides a list of links to resources on servlets and other technologies.

www.javaskyline.com
Java Skyline is an online magazine for servlet developers.

www.rfc-editor.org
The RFC Editor provides a search engine for RFCs (Request for Comments). Many of these RFCs provide details of Web-related technologies. RFCs of interest to servlet developers include URIs
SUMMARY

- The classes and interfaces used to define servlets are found in packages `javax.servlet` and `javax.servlet.http`.
- The Internet offers many protocols. The HTTP protocol (Hypertext Transfer Protocol) that forms the basis of the World Wide Web uses URIs (Uniform Resource Identifiers) to locate resources on the Internet.
- Common URLs represent files or directories and can represent complex tasks such as database lookups and Internet searches.
- JavaServer Pages technology is an extension of servlet technology.
- Servlets are normally executed as part of a Web server (also known as the servlet container).
- Servlets and JavaServer Pages have become so popular that they are now supported by most major Web servers and application servers.
- All servlets must implement the `Servlet` interface. The methods of interface `Servlet` are invoked automatically by the servlet container.
- A servlet’s life cycle begins when the servlet container loads the servlet into memory—normally in response to the first request to that servlet. Before the servlet can handle the first request, the servlet container invokes the servlet’s `init` method. After `init` completes execution, the servlet can respond to its first request. All requests are handled by a servlet’s `service` method, which may be called many times during the life cycle of a servlet. When the servlet container terminates the servlet, the servlet’s `destroy` method is called to release servlet resources.
- The servlet packages define two abstract classes that implement the interface `Servlet`—class `GenericServlet` and class `HttpServlet`. Most servlets extend one of these classes and override some or all of their methods with appropriate customized behaviors.
- The key method in every servlet is method `service`, which receives both a `ServletRequest` object and a `ServletResponse` object. These objects provide access to input and output streams that allow the servlet to read data from the client and send data to the client.
- Web-based servlets typically extend class `HttpServlet`. Class `HttpServlet` overrides method `service` to distinguish between the typical requests received from a client Web browser. The two most common HTTP request types (also known as request methods) are `get` and `post`.
- Class `HttpServlet` defines methods `doGet` and `doPost` to respond to `get` and `post` requests from a client, respectively. These methods are called by the `HttpServlet` class’s `service` method, which is called when a request arrives at the server.
- Methods `doGet` and `doPost` receive as arguments an `HttpServletRequest` object and an `HttpServletResponse` object that enable interaction between the client and the server.
- A response is sent to the client through a `PrintWriter` object returned by the `getWriter` method of the `HttpServletResponse` object.
- The `HttpServletResponse` object’s `setContentType` method specifies the MIME type of the response to the client. This enables the client browser to understand and handle the content.
The server **localhost** (IP address **127.0.0.1**) is a well-known server host name on most computers that support TCP/IP-based networking protocols such as HTTP. This host name can be used to test TCP/IP applications on the local computer.

The Tomcat server awaits requests from clients on port 8080. This port number must be specified as part of the URL to request a servlet running in Tomcat.

The client can access a servlet only if that servlet is installed on a server that can respond to servlet requests. Web servers that support servlets normally have an installation procedure for servlets.

Tomcat is a fully functional implementation of the JSP and servlet standards. It includes a Web server, so it can be used as a stand-alone test container for JSPs and servlets.

Tomcat can be specified as the handler for JSP and servlet requests received by popular Web servers such as Apache and IIS. Tomcat also is integrated into the Java 2 Enterprise Edition reference implementation from Sun Microsystems.

JSPs, servlets and their supporting files are deployed as part of Web applications. In Tomcat, Web applications are deployed in the **webapps** subdirectory of the Tomcat installation.

A Web application has a well-known directory structure in which all the files that are part of the application reside. This directory structure can be set up by the Tomcat server administrator in the **webapps** directory, or the entire directory structure can be archived in a Web application archive file. Such an archive is known as a WAR file and ends with the **.war** file extension.

If a WAR file is placed in the **webapps** directory, when the Tomcat server starts up it extracts the contents of the WAR file into the appropriate **webapps** subdirectory structure.

The Web application directory structure is separated into a context root—the top-level directory for an entire Web application—and several subdirectories. The context root is the root directory for the Web application. All the JSPs, HTML documents, servlets and supporting files such as images and class files reside in this directory or its subdirectories. The **WEB-INF** directory contains the Web application deployment descriptor (**web.xml**). The **WEB-INF/classes** directory contains the servlet class files and other supporting class files used in a Web application. The **WEB-INF/lib** directory contains Java archive (JAR) files that may include servlet class files and other supporting class files used in a Web application.

Before deploying a Web application, the servlet container must be made aware of the context root for the Web application. In Tomcat, this can be done simply by placing a directory in the **webapps** subdirectory. Tomcat uses the directory name as the context name.

Deploying a Web application requires the creation of a deployment descriptor (**web.xml**).

HTTP **get** requests can be typed directly into your browser’s **Address** or **Location** field.

Parameters are passed as name/value pairs in a **get** request. A ? separates the URL from the data passed as part of a **get** request. Name/value pairs are passed with the name and the value separated by =. If there is more than one name/value pair, each name/value pair is separated by &.

Method **getParameter** of interface **HttpServletRequest** receives the parameter name as an argument and returns the corresponding **String** value, or **null** if the parameter is not part of the request.

An HTTP **post** request is often used to post data from an Web-page form to a server-side form handler that processes the data.

Browsers often cache (save on disk) Web pages so they can quickly reload the pages. Browsers do not cache the server’s response to a **post** request.

Method **doPost** receives the same two arguments as **doGet**—an object that implements interface **HttpServletRequest** to represent the client’s request and an object that implements interface **HttpServletResponse** to represent the servlet’s response.
Method `sendRedirect` of `HttpServletResponse` redirects a request to the specified URL.

- When a servlet uses a relative path to reference another static or dynamic resource, the servlet assumes the same context root unless a complete URL is specified for the resource.

- Once method `sendRedirect` executes, processing of the request by the servlet that called `sendRedirect` terminates.

- When redirecting requests, the request parameters from the original request are passed as parameters to the new request. Additional request parameters also can be passed.

- New parameters are added to the existing request parameters. If a new parameter has the same name as an existing parameter, the new parameter value takes precedence over the original value. However, all the values are still passed.

- The complete set of values for a given request-parameter name can be obtained by calling method `getParameterValues` from interface `HttpServletRequest`, which receives the parameter name as an argument and returns an array of `Strings` containing the parameter values in order from the most recently added value for that parameter to the least recently added.

- Many Web sites today provide custom Web pages and/or functionality on a client-by-client basis.

- HTTP is a stateless protocol—it does not support persistent information that could help a Web server determine that a request is from a particular client.

- Cookies can store information on the user’s computer for retrieval later in the same or in future browsing sessions.

- Cookies are text-based data that are sent by servlets (or other similar technologies) as part of responses to clients.

- Every HTTP-based interaction between a client and a server includes a header containing information about the request (when the communication is from the client to the server) or information about the response (when the communication is from the server to the client).

- When the server receives a request, the header includes information such as the request type (e.g., `get` or `post`) and the cookies stored on the client machine by the server.

- When the server formulates its response, the header information includes any cookies the server wants to store on the client computer and other information such as the MIME type of the response.

- Depending on the maximum age of a cookie, the Web browser either maintains the cookie for the duration of the browsing session or stores the cookie on the client computer for future use. When the browser requests a resource from a server, cookies previously sent to the client by that server are returned to the server as part of the request formulated by the browser. Cookies are deleted automatically when they expire.

- By default, cookies only exist for the current browsing session (until the user closes the browser). To make cookies persist beyond the current session, call `Cookie` method `setMaxAge` to indicate the number of seconds until the cookie expires.

- Method `addCookie` of interface `HttpServletResponse` adds a cookie to the response. Cookies are sent to the client as part of the HTTP header. The header information is always provided to the client first, so the cookies should be added before the response is output.

- `HttpServletRequest` method `getCookies` returns an array of `Cookie` objects. Method `getCookies` returns `null` if there are no cookies in the request.

- `Cookie` method `getName` retrieves the name of a cookie. `Cookie` method `getValue` retrieves the value of a cookie.

- Java provides enhanced session An alternative approach to cookies is to track a session with `HttpSessions`, which eliminate the problems associated with clients disabling cookies in their browsers by making the session-tracking mechanism transparent to the programmer.
• Method `getSession` of interface `HttpServletRequest` obtains an `HttpSession` object for the client.

• Like a cookie, an `HttpSession` object can store name/value pairs. In sessions, these are called attributes, and they are stored with `setAttribute` and retrieved with `getAttribute`.

• Name/value pairs added to an `HttpSession` object with `setAttribute` remain available until the client’s current browsing session ends or until the session is explicitly invalidated by a call to the `HttpSession` object’s `invalidate` method.

• `HttpSession` method `getID` obtains the session’s unique ID number.

• `HttpSession` method `isNew` determines whether a session is new or already exists. Method `getCreationTime` obtains the time at which the session was created.

• `HttpSession` method `getLastAccessedTime` obtains the time at which the session was last accessed.

• `HttpSession` method `getMaxInactiveInterval` obtains the maximum amount of time that an `HttpSession` object can be inactive before the servlet container discards it.

• Many of today’s applications are three-tier distributed applications, consisting of a user interface, business logic and database access.

• In multi-tier architectures, Web servers often represent the middle tier. They provide the business logic that manipulates data from databases and that communicates with client Web browsers.

• `Servlet` method `init` takes a `ServletConfig` argument and throws a `ServletException`. The argument provides the servlet with information about its initialization parameters that are specified in a `servlet` element in the deployment descriptor. Each parameter appears in an `init-param` element with child elements `param-name` and `param-value`.

**TERMINOLOGY**

- `addCookie` method of `HttpServletResponse`
- Apache Tomcat server
- cache a Web page
- commit a response
- context root
- `Cookie` class
- `delete` request
- deploy a Web application
- deployment descriptor
- `destroy` method of `Servlet`
- `doGet` method of `HttpServletRequest`
- `doPost` method of `HttpServletRequest`
- `GenericServlet` class from `javax.servlet`
- `get` request
- `getAttribute` method of `HttpSession`
- `getAttributeNames` method of `HttpServletRequest`
- `getCookies` method of `HttpServletRequest`
- `getCreationTime` method of `HttpServletRequest`
- `getID` method of `HttpSession`
- `getMaxInactiveInterval` method of `HttpSession`
- `getName` method of `Cookie`
- `getOutputStream` method of `HttpServletResponse`
- `getParameter` method of `HttpServletRequest`
- `getParameterNames` method of `HttpServletRequest`
- `getParameterValues` method of `HttpServletRequest`
- `getServletConfig` method of `Servlet`
- `getServletInfo` method of `Servlet`
- `getServletRequest` method of `HttpServletRequest`
- `getValue` method of `Cookie`
- `getWriter` method of `HttpServletResponse`
- host name
- HTTP (Hypertext Transfer Protocol)
- HTTP header
HTTP request
HttpServletRequest interface
HttpServletResponse interface
HttpSession interface
init method of Servlet
invalidate method of HttpSession
isNew method of HttpSession

9.1 Fill in the blanks in each of the following statements:

a) Classes HttpServlet and GenericServlet implement the ________ interface.
b) Class HttpServlet defines the methods ________ and ________ to respond to get and post requests from a client.
c) HttpServletResponse method ________ obtains a character-based output stream that enables text data to be sent to the client.
d) The form attribute ________ specifies the server-side form handler, i.e., the program that handles the request.
e) ________ is the well-known host name that refers to your own computer.
f) Cookie method ________ returns a String the name of the cookie as set with ________ or the constructor.
g) HttpServletRequest method getSession returns an ________ object for the client.

9.2 State whether each of the following is true or false. If false, explain why.

a) Servlets usually are used on the client side of a networking application.
b) Servlet methods are executed automatically.
c) The two most common HTTP requests are get and put.
d) The well-known port number for Web requests is 55.
e) Cookies never expire.
f) HttpSession expire only when the browsing session ends or when the invalidate method is called.
g) The HttpSession method getAttribute returns the object associated with a particular name.

ANSWERS TO SELF-REVIEW EXERCISES

9.1 a) Servlet. b) doGet, doPost. c) getWriter. d) action. e) localhost. f) getName, setName. g) HttpSession.

9.2 a) False. Servlets are usually used on the server side.
b) True.
c) False. The two most common HTTP request types are get and post.
d) False. The well-known port number for Web requests is 80.
e) False. Cookies expire when they reach their maximum age.
f) True.
g) True.

EXERCISES

9.3 Modify the Cookie example of Fig. 9.20 to list prices for each book in the book recommendations. Also, allow the user to select some or all of the recommended books and “order” them. Deploy your Web application on the Tomcat server.

9.4 Modify the HttpSession example of Fig. 9.24 to list prices for each book in the book recommendations. Also, allow the user to select some or all of the recommended books and “order” them. Deploy your Web application on the Tomcat server.

9.5 Create a Web application for dynamic FAQs. The application should obtain the information to create the dynamic FAQ Web page from a database that consists of a Topics table and an FAQ table. The Topics table should have two fields—a unique integer ID for each topic (topicID) and a name for each topic (topicName). The FAQ table should have three fields—the topicID (a foreign key), a string representing the question (question) and the answer to the question (answer). When the servlet is invoked, it should read the data from the database and return a dynamically created Web page containing each question and answer, sorted by topic.

9.6 Modify the Web application of Exercise 9.5 so that the initial request to the servlet returns a Web page of topics in the FAQ database. Then, the user can hyperlink to another servlet that returns only the frequently asked questions for a particular topic.

9.7 Modify the Web application of Fig. 9.27 to allow the user to see the survey results without responding to the survey.

9.8 Fig. 9.27 would allow users to vote as many times as they want by simply returning to the survey Web page and submitting additional votes. Modify your solution to Exercise 9.7 such that it uses cookies that last for one day to prevent users from voting more than once a day. When a user returns to the site, the cookie previously stored on their system is sent to the server. The servlet should check for the cookie and, if it exists, indicate that the client already voted in the last 24 hours. The servlet should also return the current survey results.

9.9 Modify the Web application of Fig. 9.27 to make it generic for use with any survey of the appropriate form. Use servlet parameters (as discussed in Section 9.8) to specify the survey options. When the user requests the survey, dynamically generate a form containing the survey options. Deploy this Web application twice using different context roots. Note: You may need to modify the database in this example so that it can store multiple surveys at once.
9.10 Write a Web application that consists of a servlet (DirectoryServlet) and several Web documents. Document index.html should be the first document the user sees. In that document, you should have a series of hyperlinks for other Web pages in your site. When clicked, each hyperlink should invoke the servlet with a get request that contains a page parameter. The servlet should obtain parameter page and redirect the request to the appropriate document.

9.11 Modify the Web application of Exercise 9.10 so that the first document the user sees in the browser is dynamically generated from servlet initialization parameters (as discussed in Section 9.8) by servlet HomePageServlet. There should be a separate initialization parameter for each page in the Web site. The HomePageServlet reads each parameter name and value and creates a HashMap of the parameter name/value pairs. This information should be used to create the initial home page dynamically. The HashMap also should be placed in the ServletContext with method setAttribute, so that the HashMap can be used in the DirectoryServlet to determine where to direct each request. The dynamic home page should have hyperlinks to each document in the Web site. As in Exercise 9.10, when the user clicks a link, servlet DirectoryServlet should be invoked and passed a page parameter. Then, the DirectoryServlet should obtain the HashMap from the ServletContext, look up the corresponding document and redirect the user to that document.
Objectives

• To be able to create and deploy JavaServer Pages.
• To use JSP’s implicit objects and Java to create dynamic Web pages.
• To specify global JSP information with directives.
• To use actions to manipulate JavaBeans in a JSP, to include resources dynamically and to forward requests to other JSPs.
• To create custom tag libraries that encapsulate complex functionality in new tags that can be reused by JSP programmers and Web-page designers.

A tomato does not communicate with a tomato, we believe. We could be wrong.
Gustav Eckstein

A donkey appears to me like a horse translated into Dutch.
Georg Christoph Lichtenberg

Talent is a question of quantity. Talent does not write one page: it writes three hundred.
Jules Renard

Every action must be due to one or other of seven causes: chance, nature, compulsion, habit, reasoning, anger, or appetite.
Aristotle
10.1 Introduction

Our discussion of client–server networking continues in this chapter with JavaServer Pages (JSP)—an extension of servlet technology. JavaServer Pages simplify the delivery of dynamic Web content. They enable Web application programmers to create dynamic content by reusing predefined components and by interacting with components using server-side scripting. JavaServer Page programmers can reuse JavaBeans and create custom tag libraries that encapsulate complex, dynamic functionality. Custom-tag libraries even enable Web-page designers who are not familiar with Java to enhance Web pages with powerful dynamic content and processing capabilities.

In addition to the classes and interfaces for programming servlets (from packages `javax.servlet` and `javax.servlet.http`), classes and interfaces specific to JavaServer Pages programming are located in packages `javax.servlet.jsp` and `javax.servlet.jsp.tagext`. We discuss many of these classes and interfaces throughout this chapter as we present JavaServer Pages fundamentals. For a complete description of JavaServer Pages, see the JavaServer Pages 1.1 specification, which can be downloaded from `java.sun.com/products/jsp/download.html`. We also
include other JSP resources in Section 10.9. [Note: The source code and images for all the examples in this chapter can be found on the CD that accompanies this book and on our Web site www.deitel.com.]

10.2 JavaServer Pages Overview

There are four key components to JSPs: directives, actions, scriptlets and tag libraries. Directives are messages to the JSP container that enable the programmer to specify page settings, to include content from other resources and to specify custom tag libraries for use in a JSP. Actions encapsulate functionality in predefined tags that programmers can embed in a JSP. Actions often are performed based on the information sent to the server as part of a particular client request. They also can create Java objects for use in JSP scriptlets. Scriptlets, or scripting elements, enable programmers to insert Java code that interacts with components in a JSP (and possibly other Web application components) to perform request processing. Tag libraries are part of the tag extension mechanism that enables programmers to create custom tags. Such tags enable programmers to manipulate JSP content. These JSP component types are discussed in detail in subsequent sections.

In many ways, Java Server Pages look like standard XHTML or XML documents. In fact, JSPs normally include XHTML or XML markup. Such markup is known as fixed-template data or fixed-template text. Fixed-template data often help a programmer decide whether to use a servlet or a JSP. Programmers tend to use JSPs when most of the content sent to the client is fixed template data and only a small portion of the content is generated dynamically with Java code. Programmers use servlets when only a small portion of the content sent to the client is fixed-template data. In fact, some servlets do not produce content. Rather, they perform a task on behalf of the client, then invoke other servlets or JSPs to provide a response. Note that in most cases, servlet and JSP technologies are interchangeable. As with servlets, JSPs normally execute as part of a Web server. The server often is referred to as the JSP container.

Software Engineering Observation 10.1

Literal text in a JSP becomes string literals in the servlet that represents the translated JSP.

When a JSP-enabled server receives the first request for a JSP, the JSP container translates that JSP into a Java servlet that handles the current request and future requests to the JSP. If there are any errors compiling the new servlet, these errors result in translation-time errors. The JSP container places the Java statements that implement the JSP’s response in method _jspService at translation time. If the new servlet compiles properly, the JSP container invokes method _jspService to process the request. The JSP may respond directly to the request or may invoke other Web application components to assist in processing the request. Any errors that occur during request processing are known as request-time errors.

Performance Tip 10.1

Some JSP containers translate JSPs to servlets at installation time. This eliminates the translation overhead for the first client that requests each JSP.

Overall, the request/response mechanism and life cycle of a JSP is the same as that of a servlet. JSPs can define methods jspInit and jspDestroy (similar to servlet
methods **init** and **destroy**), which the JSP container invokes when initializing a JSP and terminating a JSP, respectively. JSP programmers can define these methods using JSP declarations—part of the JSP scripting mechanism.

### 10.3 A First JavaServer Page Example

We begin our introduction to JavaServer Pages with a simple example (Fig. 10.1) in which the current date and time are inserted into a Web page using a JSP expression.

As you can see, most of **clock.jsp** consists of XHTML markup. In cases like this, JSPs are easier to implement than servlets. In a servlet that performs the same task as this JSP, each line of XHTML markup typically is a separate Java statement that outputs the string representing the markup as part of the response to the client. Writing code to output markup can often lead to errors. Most JSP editors provide syntax coloring to help programmers check that their markup follows proper syntax.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.1: clock.jsp -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv = "refresh" content = "60" />
<title>A Simple JSP Example</title>
<style type = "text/css">
.big { font-family: helvetica, arial, sans-serif;
font-weight: bold;
font-size: 2em; }
</style>
</head>
<body>
<p class = "big">Simple JSP Example</p>
<table style = "border: 6px outset;">
<tr>
<td style = "background-color: black;">
<p class = "big" style = "color: cyan;">
<!-- JSP expression to insert date/time -->
<%= new java.util.Date() %>
</p>
</td>
</tr>
</table>
</body>
```

Fig. 10.1 Using a JSP expression to insert the date and time in a Web page (part 1 of 2).
Software Engineering Observation 10.2

JavaServer Pages are easier to implement than servlets when the response to a client request consists primarily of markup that remains constant between requests.

The JSP of Fig. 10.1 generates an XHTML document that displays the current date and time. The key line in this JSP (line 30) is the expression

```jsp
<%= new java.util.Date() %>
```

JSP expressions are delimited by `<%=` and `%>`. This particular expression creates a new instance of class `Date` from package `java.util`. When the client requests this JSP, the preceding expression inserts the `String` representation of the date and time in the response to the client. [Note: Proper internationalization requires that the JSP return the date in the client locale’s format. In this example, the server’s local determines the `String` representation of the `Date`. In Fig. 10.9, `clock2.jsp` demonstrates how to determine the client’s locale and uses a `DateFormat` (package `java.text`) object to format the date using that locale.]

Software Engineering Observation 10.3

The JSP container converts the result of every JSP expression into a `String` that is output as part of the response to the client.
Note that we use the XHTML meta element on line 10 to set a refresh interval of 60 seconds for the document. This causes the browser to request clock.jsp every 60 seconds. For each request to clock.jsp, the JSP container reevaluates the expression on line 30, creating a new Date object with the server’s current date and time.

As in Chapter 9, we use Apache Tomcat to test our JSPs in the advjhtp1 Web application we created previously. For details on creating and configuring the advjhtp1 Web application, review Section 9.3.1 and Section 9.3.2. To test clock.jsp, create a new directory called jsp in the advjhtp1 subdirectory of Tomcat’s webapps directory. Next, copy clock.jsp into the jsp directory. Open your Web browser and enter the following URL to test clock.jsp:

http://localhost:8080/advjhtp1/jsp/clock.jsp

When you first invoke the JSP, notice the delay as Tomcat translates the JSP into a servlet and invokes the servlet to respond to your request. [Note: It is not necessary to create a directory named jsp in a Web application. We use this directory to separate the examples in this chapter from the servlet examples in Chapter 9.]

### 10.4 Implicit Objects

Implicit objects provide programmers with access to many servlet capabilities in the context of a JavaServer Page. Implicit objects have four scopes: application, page, request and session. The JSP and servlet container application owns objects with application scope. Any servlet or JSP can manipulate such objects. Objects with page scope exist only in the page that defines them. Each page has its own instances of the page-scope implicit objects. Objects with request scope exist for the duration of the request. For example, a JSP can partially process a request, then forward the request to another servlet or JSP for further processing. Request-scope objects go out of scope when request processing completes with a response to the client. Objects with session scope exist for the client’s entire browsing session. Figure 10.2 describes the JSP implicit objects and their scopes. This chapter demonstrates several of these objects.

<table>
<thead>
<tr>
<th>Implicit Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Scope</strong></td>
<td></td>
</tr>
<tr>
<td>application</td>
<td>This javax.servlet.ServletContext object represents the container in which the JSP executes.</td>
</tr>
<tr>
<td><strong>Page Scope</strong></td>
<td></td>
</tr>
<tr>
<td>config</td>
<td>This javax.servlet.ServletConfig object represents the JSP configuration options. As with servlets, configuration options can be specified in a Web application descriptor.</td>
</tr>
<tr>
<td>exception</td>
<td>This java.lang.Throwale object represents the exception that is passed to the JSP error page. This object is available only in a JSP error page.</td>
</tr>
</tbody>
</table>

Fig. 10.2 JSP implicit objects (part 1 of 2).
Note that many of the implicit objects extend classes or implement interfaces discussed in Chapter 9. Thus, JSPs can use the same methods that servlets use to interact with such objects, as described in Chapter 9. Most of the examples in this chapter use one or more of the implicit objects in Fig. 10.2.

### 10.5 Scripting

JavaServer Pages often present dynamically generated content as part of an XHTML document sent to the client in response to a request. In some cases, the content is static, but is output only if certain conditions are met during a request (such as providing values in a `form` that submits a request). JSP programmers can insert Java code and logic in a JSP using scripting.

**Software Engineering Observation 10.4**

JavaServer Pages currently support scripting only with Java. Future JSP versions may support other scripting languages.
10.5.1 Scripting Components

JSP scripting components include scriptlets, comments, expressions, declarations and escape sequences. This section describes each of these scripting components. Many of these scripting components are demonstrated in Fig. 10.4 at the end of Section 10.5.2.

Scriptlets are blocks of code delimited by `<%` and `%>`. They contain Java statements that the container places in method _jspService at translation time.

JSPs support three comment styles: JSP comments, XHTML comments and comments from the scripting language. JSP comments are delimited by `<%--` and `-->`. Such comments can be placed throughout a JSP, but not inside scriptlets. XHTML comments are delimited with `<!--` and `-->`. These comments can be placed throughout a JSP, but not inside scriptlets. Scripting language comments are currently Java comments, because Java is the only JSP scripting language at the present time. Scriptlets can use Java’s single-line comments (delimited by `/` and `/`) and multiline comments (delimited by `/*` and `*/`).

**Common Programming Error 10.1**

Placing a JSP comment or XHTML comment inside a scriptlet is a translation-time syntax error that prevents the JSP from being translated properly.

JSP comments and scripting-language comments are ignored and do not appear in the response to a client. When clients view the source code of a JSP response, they will see only the XHTML comments in the source code. The different comment styles are useful for separating comments that the user should be able to see from comments that document logic processed on the server.

A JSP expression, delimited by `<%=` and `%>`, contains a Java expression that is evaluated when a client requests the JSP containing the expression. The container converts the result of a JSP expression to a `String` object, then outputs the `String` as part of the response to the client.

Declarations (delimited by `<%!` and `%>`) enable a JSP programmer to define variables and methods. Variables become instance variables of the servlet class that represents the translated JSP. Similarly, methods become members of the class that represents the translated JSP. Declarations of variables and methods in a JSP use Java syntax. Thus, a variable declaration must end in a semicolon, as in

```
<%! int counter = 0; %>
```

**Common Programming Error 10.2**

Declaring a variable without using a terminating semicolon is a syntax error.

**Software Engineering Observation 10.5**

Variables and methods declared in JSP declarations are initialized when the JSP is initialized and are available for use in all scriptlets and expressions in that JSP. Variables declared in this manner become instance variables of the servlet class that represents the translated JSP.

**Software Engineering Observation 10.6**

As with servlets, JSPs should not store client state information in instance variables. Rather, JSPs should use the JSP implicit `session` object.
Special characters or character sequences that the JSP container normally uses to delimit JSP code can be included in a JSP as literal characters in scripting elements, fixed template data and attribute values using *escape sequences*. Figure 10.3 shows the literal character or characters and the corresponding escape sequences and discusses where to use the escape sequences.

### 10.5.2 Scripting Example

The JSP of Fig. 10.4 demonstrates basic scripting capabilities by responding to *get* requests. The JSP enables the user to input a first name, then outputs that name as part of the response. Using scripting, the JSP determines whether a *firstName* parameter was passed to the JSP as part of the request; if not, the JSP returns an XHTML document containing a *form* through which the user can input a first name. Otherwise, the JSP obtains the *firstName* value and uses it as part of an XHTML document that welcomes the user to JavaServer Pages.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;%</code></td>
<td><code>&lt;\%</code></td>
<td>The character sequence <code>&lt;%</code> normally indicates the beginning of a scriptlet. The <code>&lt;\%</code> escape sequence places the literal characters <code>&lt;%</code> in the response to the client.</td>
</tr>
<tr>
<td><code>%&gt;</code></td>
<td><code>%\&gt;</code></td>
<td>The character sequence <code>%&gt;</code> normally indicates the end of a scriptlet. The <code>%\&gt;</code> escape sequence places the literal characters <code>&gt;</code> in the response to the client.</td>
</tr>
<tr>
<td><code>'</code></td>
<td><code>'</code></td>
<td>As with string literals in a Java program, the escape sequences for characters <code>,</code>, <code>&quot;</code> and <code>\</code> allow these characters to appear in attribute values. Remember that the literal text in a JSP becomes string literals in the servlet that represents the translated JSP.</td>
</tr>
</tbody>
</table>

**Fig. 10.3**  JSP escape sequences.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- Fig. 10.4: welcome.jsp -->
<!-- JSP that processes a "get" request containing data. -->
<html xmlns = "http://www.w3.org/1999/xhtml">
    <!-- head section of document -->
    <head>
        <title>Processing "get" requests with data</title>
    </head>
</html>
```

**Fig. 10.4**  Scripting a JavaServer Page—**welcome.jsp** (part 1 of 3).
<!-- body section of document -->
<body>
    <!-- begin scriptlet
    String name = request.getParameter("firstName");
    if ( name != null ) {
        <h1>Hello <%= name %>, <br />
            Welcome to JavaServer Pages!
        </h1>
    } // end if
    else {
        <form action = "welcome.jsp" method = "get">
            <p>Type your first name and press Submit</p>
            <p><input type = "text" name = "firstName" />
            <input type = "submit" value = "Submit" />
        </form>
    } // end else
</body>
</html> <!-- end XHTML document -->
Chapter 10

JavaServer Pages (JSP) 603

Notice that the majority of the code in Fig. 10.4 is XHTML markup (i.e., fixed template data). Throughout the body element are several scriptlets (lines 17–23, 30–35 and 45–49) and a JSP expression (line 26). Note that three comment styles appear in this JSP.

The scriptlets define an if/else structure that determines whether the JSP received a value for the first name as part of the request. Line 19 uses method getParameter of JSP implicit object request (an HttpServletRequest object) to obtain the value for parameter firstName and assigns the result to variable name. Line 21 determines if name is not null, (i.e., a value for the first name was passed to the JSP as part of the request). If this condition is true, the scriptlet terminates temporarily so the fixed template data at lines 25–28 can be output. The JSP expression in line 26 outputs the value of variable name (i.e., the first name passed to the JSP as a request parameter. The scriptlet continues at lines 30–35 with the closing curly brace of the if structure’s body and the beginning of the else part of the if/else structure. If the condition at line 21 is false, lines 25–28 are not output. Instead, lines 37–43 output a form element. The user can type a first name in the form and press the Submit button to request the JSP again and execute the if structure’s body (lines 25–28).

Software Engineering Observation 10.7

Scriptlets, expressions and fixed template data can be intermixed in a JSP to create different responses based on information in a request to a JSP.

Testing and Debugging Tip 10.1

It is sometimes difficult to debug errors in a JSP, because the line numbers reported by a JSP container normally refer to the servlet that represents the translated JSP, not the original JSP line numbers. Program development environments such as Sun Microsystems, Inc.’s Forte for Java Community Edition enable JSPs to be compiled in the environment, so you can see syntax error messages. These messages include the statement in the servlet that represents the translated JSP, which can be helpful in determining the error.

Testing and Debugging Tip 10.2

Many JSP containers store the servlets representing the translated JSPs. For example, the Tomcat installation directory contains a subdirectory called work in which you can find the source code for the servlets translated by Tomcat.
To test Fig. 10.4 in Tomcat, copy `welcome.jsp` into the `jsp` directory created in Section 10.3. Open your Web browser and enter the following URL to test `welcome.jsp`:

```http://localhost:8080/advjhtp1/jsp/welcome.jsp```

When you first execute the JSP, it displays the form in which you can enter your first name, because the preceding URL does not pass a `firstName` parameter to the JSP. After you submit your first name, your browser should appear as shown in the second screen capture of Fig. 10.4. Note: As with servlets, it is possible to pass `get` request arguments as part of the URL. The following URL supplies the `firstName` parameter to `welcome.jsp`:

```http://localhost:8080/advjhtp1/jsp/welcome.jsp?firstName=Paul```

### 10.6 Standard Actions

We continue our JSP discussion with the *JSP standard actions* (Fig. 10.5). These actions provide JSP implementors with access to several of the most common tasks performed in a JSP, such as including content from other resources, forwarding requests to other resources and interacting with JavaBeans. JSP containers process actions at request time. Actions are delimited by `<jsp:action>` and `</jsp:action>`, where `action` is the standard action name. In cases where nothing appears between the starting and ending tags, the XML empty element syntax `<jsp:action />` can be used. Figure 10.5 summarizes the JSP standard actions. We use the actions in the next several subsections.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;jsp:include&gt;</code></td>
<td>Dynamically includes another resource in a JSP. As the JSP executes, the referenced resource is included and processed.</td>
</tr>
<tr>
<td><code>&lt;jsp:forward&gt;</code></td>
<td>Forwards request processing to another JSP, servlet or static page. This action terminates the current JSP’s execution.</td>
</tr>
<tr>
<td><code>&lt;jsp:plugin&gt;</code></td>
<td>Allows a plug-in component to be added to a page in the form of a browser-specific <code>object</code> or <code>embed</code> HTML element. In the case of a Java applet, this action enables the downloading and installation of the Java Plug-in, if it is not already installed on the client computer.</td>
</tr>
<tr>
<td><code>&lt;jsp:param&gt;</code></td>
<td>Used with the <code>include</code>, <code>forward</code> and <code>plugin</code> actions to specify additional name/value pairs of information for use by these actions.</td>
</tr>
</tbody>
</table>

**JavaBean Manipulation**

| `<jsp:useBean>` | Specifies that the JSP uses a JavaBean instance. This action specifies the scope of the bean and assigns it an ID that scripting components can use to manipulate the bean. |

---

*Fig. 10.5* JSP standard actions (part 1 of 2).
10.6.1 <jsp:include> Action

JavaServer Pages support two include mechanisms—the <jsp:include> action and the include directive. Action <jsp:include> enables dynamic content to be included in a JavaServer Page. If the included resource changes between requests, the next request to the JSP containing the <jsp:include> action includes the new content of the resource. On the other hand, the include directive copies the content into the JSP once, at JSP translation time. If the included resource changes, the new content will not be reflected in the JSP that used the include directive unless that JSP is recompiled. Figure 10.6 describes the attributes of action <jsp:include>.

Software Engineering Observation 10.8

According to the JavaServer Pages 1.1 specification, a JSP container is allowed to determine whether a resource included with the include directive has changed. If so, the container can recompile the JSP that included the resource. However, the specification does not provide a mechanism to indicate a change in an included resource to the container.

Performance Tip 10.2

The <jsp:include> action is more flexible than the include directive, but requires more overhead when page contents change frequently. Use the <jsp:include> action only when dynamic content is necessary.

Common Programming Error 10.3

Setting the <jsp:include> action’s flush attribute to false is a translation-time error. Currently, the flush attribute supports only true values.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>Specifies the relative URI path of the resource to include. The resource must be part of the same Web application.</td>
</tr>
<tr>
<td>flush</td>
<td>Specifies whether the buffer should be flushed after the include is performed. In JSP 1.1, this attribute is required to be true.</td>
</tr>
</tbody>
</table>
Common Programming Error 10.4

Not specifying the `<jsp:include>` action’s `flush` attribute is a translation-time error. Specifying this attribute is mandatory.

Common Programming Error 10.5

Specifying in a `<jsp:include>` action a page that is not part of the same Web application is a request-time error. In such a case, the `<jsp:include>` action does not include any content.

The next example demonstrates action `<jsp:include>` using four XHTML and JSP resources that represent both static and dynamic content. JavaServer Page `include.jsp` (Fig. 10.10) includes three other resources: `banner.html` (Fig. 10.7), `toc.html` (Fig. 10.8) and `clock2.jsp` (Fig. 10.9). JavaServer Page `include.jsp` creates an XHTML document containing a table in which `banner.html` spans two columns across the top of the table, `toc.html` is the left column of the second row and `clock2.jsp` (a simplified version of Fig. 10.1) is the right column of the second row. Figure 10.10 uses three `<jsp:include>` actions (lines 38–39, 48 and 55–56) as the content in td elements of the table. Using two XHTML documents and a JSP in Fig. 10.10 demonstrates that JSPs can include both static and dynamic content. The output windows in Fig. 10.10 demonstrate the results of two separate requests to `include.jsp`.

Figure 10.9 (`clock2.jsp`) demonstrates how to determine the client’s `Locale` (package `java.util`) and uses that `Locale` to format a `Date` with a `DateFormat` (package `java.text`) object. Line 14 invokes the `request` object’s `getLocale` method, which returns the client’s `Locale`. Lines 17–20 invoke `DateFormat static` method `getDateTimeInstance` to obtain a `DateFormat` object. The first two arguments indicate that the date and time formats should each be `LONG` format (other options are `FULL`, `MEDIUM`, `SHORT` and `DEFAULT`). The third argument specifies the `Locale` for which the `DateFormat` object should format the date. Line 25 invokes the `DateFormat` object’s `format` method to produce a `String` representation of the `Date`. The `DateFormat` object formats this `String` for the `Locale` specified on lines 17–20. [Note: This example works for Western languages that use the ISO-8859-1 character set. However, for languages that do not use this character set, the JSP must specify the proper character set using the JSP `page` directive (Section 10.7.1). At the site `java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html`, Sun provides a list of character encodings. The response’s content type defines the character set to use in the response. The content type has the form: "`mime-Type; charset=encoding" (e.g., "text/html;charset=ISO-8859-1"].

To test Fig. 10.10 in Tomcat, copy `banner.html`, `toc.html`, `clock2.jsp`, `include.jsp` and the `images` directory into the `jsp` directory created in Section 10.3. Open your Web browser and enter the following URL to test `welcome.jsp`:

```
http://localhost:8080/advjhtp1/jsp/include.jsp
```

---

1  <!-- Fig. 10.7: banner.html -->
2  <!-- banner to include in another document -->

Fig. 10.7 Banner (`banner.html`) to include across the top of the XHTML document created by Fig. 10.10 (part 1 of 2).
<div style="width: 580px">
  <p>Java(TM), C, C++, Visual Basic(R), Object Technology, and Internet and World Wide Web Programming Training<br />
On-Site Seminars Delivered Worldwide</p>
  <a href="mailto:deitel@deitel.com">deitel@deitel.com</a><br />
  978.579.9911<br />
  490B Boston Post Road, Suite 200,<br />
  Sudbury, MA 01776</div>

Fig. 10.7 Banner (<code>banner.html</code>) to include across the top of the XHTML document created by Fig. 10.10 (part 2 of 2).

<!-- Fig. 10.8: toc.html -->
<!-- contents to include in another document -->
<p><a href="http://www.deitel.com/books/downloads.html">Downloads/Resources</a></p>
<p><a href="http://www.deitel.com/intro.html">Who we are</a></p>
<p>Send questions or comments about this site to <a href="mailto:deitel@deitel.com">deitel@deitel.com</a></p>

Fig. 10.8 Table of contents (<code>toc.html</code>) to include down the left side of the XHTML document created by Fig. 10.10 (part 1 of 2).
Fig. 10.9  JSP clock2.jsp to include as the main content in the XHTML document created by Fig. 10.10.

```
<!-- Fig. 10.9: clock2.jsp -->
<!-- date and time to include in another document -->
<table>
<tr>
<td style = "background-color: black;">
<p class = "big" style = "color: cyan; font-size: 3em; font-weight: bold;">
   <!-- script to determine client local and -->
   <!-- format date accordingly -->
   <%
   // get client locale
   java.util.Locale locale = request.getLocale();
   // get DateFormat for client's Locale
   java.text.DateFormat dateFormat =
      java.text.DateFormat.getDateTimeInstance(
         java.text.DateFormat.LONG,
         java.text.DateFormat.LONG, locale );
   %>
   <!-- end script -->
   <%
   // output date -->
   <%= dateFormat.format( new java.util.Date() ) %>
   </p>
</td>
</tr>
</table>
```

Fig. 10.8  Table of contents (toc.html) to include down the left side of the XHTML document created by Fig. 10.10 (part 2 of 2).

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- Fig. 10.7: include.jsp -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Using jsp:include</title>
</head>
```
includes resources with `<jsp:include>` (part 2 of 3).
10.6.2 `<jsp:forward>` Action

Action `<jsp:forward>` enables a JSP to forward request processing to a different resource. Request processing by the original JSP terminates as soon as the JSP forwards the request. Action `<jsp:forward>` has only a `page` attribute that specifies the relative URI of the resource (in the same Web application) to which the request should be forwarded.

**Software Engineering Observation 10.9**

When using the `<jsp:forward>` action, the resource to which the request will be forwarded must be in the same context (Web application) as the JSP that originally received the request.

JavaServer Page `forward1.jsp` (Fig. 10.11) is a modified version of `welcome.jsp` (Fig. 10.4). The primary difference is in lines 22–25 in which JavaServer Page `forward1.jsp` forwards the request to JavaServer Page `forward2.jsp` (Fig. 10.12). Notice the `<jsp:param>` action in lines 23–24. This action adds a request parameter representing the date and time at which the initial request was received to the request object that is forwarded to `forward2.jsp`.

The `<jsp:param>` action specifies name/value pairs of information that are passed to the `<jsp:include>`, `<jsp:forward>` and `<jsp:plugin>` actions. Every `<jsp:param>` action has two required attributes: `name` and `value`. If a `<jsp:param>` action specifies a parameter that already exists in the request, the new value for the parameter takes precedence over the original value. All values for that parameter can be obtained by using the JSP implicit object `request`'s `getParameterValues` method, which returns an array of `Strings`. 
JSP forward2.jsp uses the name specified in the <jsp:param> action ("date") to obtain the date and time. It also uses the firstName parameter originally passed to forward1.jsp to obtain the user’s first name. JSP expressions in Fig. 10.12 (lines 23 and 31) insert the request parameter values in the response to the client. The screen capture in Fig. 10.11 shows the initial interaction with the client. The screen capture in Fig. 10.12 shows the results returned to the client after the request was forwarded to forward2.jsp.

To test Fig. 10.11 and Fig. 10.12 in Tomcat, copy forward1.jsp and forward2.jsp into the jsp directory created in Section 10.3. Open your Web browser and enter the following URL to test welcome.jsp:

http://localhost:8080/advjhtp1/jsp/forward1.jsp

Fig. 10.11 JSP forward1.jsp receives a firstName parameter, adds a date to the request parameters and forwards the request to forward2.jsp for further processing (part 1 of 2).
<p><input type = "text" name = "firstName" />
<input type = "submit" value = "Submit" /></p>
</form>

<% // continue scriptlet
    } // end else
%> <%-- end scriptlet --%>
</body>
</html> <!-- end XHTML document -->

Fig. 10.11 JSP forward2.jsp receives a firstName parameter, adds a date to the request parameters and forwards the request to forward2.jsp for further processing (part 2 of 2).

<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- forward2.jsp -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
    <title>Processing a forwarded request</title>
    <style type = "text/css">
        .big {
            font-family: tahoma, helvetica, arial, sans-serif;
            font-weight: bold;
            font-size: 2em;
        }
    </style>
</head>

Fig. 10.12 JSP forward2.jsp receives a request (from forward1.jsp in this example) and uses the request parameters as part of the response to the client (part 1 of 2).
10.6.3 <jsp:plugin> Action

Action <jsp:plugin> adds an applet or JavaBean to a Web page in the form of a browser-specific object or embed XHTML element. This action also enables the client to download and install the Java Plug-in if it is not already installed. Figure 10.13 describes the attributes of action <jsp:plugin>.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Component type—bean or applet.</td>
</tr>
<tr>
<td>code</td>
<td>Class that represents the component.</td>
</tr>
</tbody>
</table>

![Image of Hello Paul, Your request was received and forwarded at Sun Jul 22 17:18:48 EDT 2001]

Fig. 10.12 JSP forward2.jsp receives a request (from forward1.jsp in this example) and uses the request parameters as part of the response to the client (part 2 of 2).
Figure 10.14 defines an applet that draws a picture using the Java2D API. The applet has three parameters that enable the JSP implementor to specify the background color for the drawing. The parameters represent the red, green and blue portions of an RGB color with values in the range 0–255. The applet obtains the parameter values in lines 21–23. If any exceptions occur while processing the parameters, the exceptions are caught at line 32 and ignored, leaving the applet with its default white background color.

```java
// Fig. 10.14: ShapesApplet.java
// Applet that demonstrates a Java2D GeneralPath.
package com.deitel.advjhtp1.jsp.applet;

// Java core packages
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
import java.awt.geom.*;

// Java extension packages
import javax.swing.*;

public class ShapesApplet extends JApplet {

Fig. 10.14  An applet to demonstrate <jsp:plugin> in Fig. 10.15 (part 1 of 3).
```
// initialize the applet
public void init()
{
  // obtain color parameters from XHTML file
  try {
    int red = Integer.parseInt( getParameter( "red" ) );
    int green = Integer.parseInt( getParameter( "green" ) );
    int blue = Integer.parseInt( getParameter( "blue" ) );

    Color backgroundColor = new Color( red, green, blue );

    setBackground( backgroundColor );
  }
  
  // if there is an exception while processing the color
  // parameters, catch it and ignore it
  catch ( Exception exception ) {
    // do nothing
  }
}

public void paint( Graphics g )
{
  // create arrays of x and y coordinates
  int xPoints[] =
  { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
  int yPoints[] =
  { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };

  // obtain reference to a Graphics2D object
  Graphics2D g2d = ( Graphics2D ) g;

  // create a star from a series of points
  GeneralPath star = new GeneralPath();

  // set the initial coordinate of the GeneralPath
  star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );

  // create the star--this does not draw the star
  for ( int k = 1; k < xPoints.length; k++ )
    star.lineTo( xPoints[ k ], yPoints[ k ] );

  // close the shape
  star.closePath();

  // translate the origin to (200, 200)
  g2d.translate( 200, 200 );

  // rotate around origin and draw stars in random colors
  for ( int j = 1; j <= 20; j++ ) {
    g2d.rotate( Math.PI / 10.0 );
  }

  // do nothing
}
Most Web browsers in use today do not support applets written for the Java 2 platform. Executing such applets in most of today’s browsers requires the Java Plug-in. Figure 10.15 uses the \texttt{<jsp:plugin>} action (lines 10–22) to embed the Java Plug-in. Line 11 indicates the package name and class name of the applet class. Line 12 indicates the code-base from which the applet should be downloaded. Line 13 indicates that the applet should be 400 pixels wide and line 14 indicates that the applet should be 400 pixels tall. Lines 16–20 specify the applet parameters. You can change the background color in the applet by changing the red, green and blue values. Note that the \texttt{<jsp:plugin>} action requires any \texttt{<jsp:param>} actions to appear in a \texttt{<jsp:params>} action.

To test the \texttt{<jsp:plugin>} action in Tomcat, copy \texttt{plugin.jsp} and \texttt{ShapesApplet.class} into the \texttt{jsp} directory created in Section 10.3. [Note: \texttt{ShapesApplet} is defined in package \texttt{com.deitel.advjhtp1.jsp.applet}. This example will work only if the proper package directory structure is defined in the \texttt{classes} directory.] Open your Web browser and enter the following URL to test \texttt{plugin.jsp}:

\texttt{http://localhost:8080/advjhtp1/jsp/plugin.jsp}

The screen captures in Fig. 10.15 show the applet executing in Microsoft Internet Explorer 5.5 and Netscape Navigator 6.0.

\begin{Verbatim}
1 \texttt{<!-- Fig. 10.15: plugin.jsp -->}
2
3 \texttt{<html>}
4
5 \texttt{<head>}
6 \texttt{<title>Using jsp:plugin to load an applet</title>}
7 \texttt{</head>}
8
9 \texttt{<body>}
10 \texttt{<jsp:plugin type = "applet"}
11 \texttt{\hspace{1em} code = "com.deitel.advjhtp1.jsp.applet.ShapesApplet"}
12 \texttt{\hspace{1em} codebase = "/advjhtp1/jsp"}
13 \texttt{\hspace{1em} width = "400"}
14 \texttt{\hspace{1em} height = "400">}
15
16 \texttt{<jsp:params>}
17 \texttt{\hspace{1em} <jsp:param name = "red" value = "255" />}
18 \texttt{</jsp:params>}
19 \texttt{</jsp:plugin>}
20 \texttt{</body>}
21 \texttt{</html>}
\end{Verbatim}
10.6.4 <jsp:useBean> Action

Action <jsp:useBean> enables a JSP to manipulate a Java object. This action creates a Java object or locates an existing object for use in the JSP. Figure 10.16 summarizes action <jsp:useBean>’s attributes. If attributes class and beanName are not specified, the JSP container attempts to locate an existing object of the type specified in attribute type. Like JSP implicit objects, objects specified with action <jsp:useBean> have page, request, session or application scope that indicates where they can be used in a Web application. Objects with page scope are accessible only to the page in which they are defined. Multiple JSP pages potentially can access objects with other scopes. For example, all JSPs that process a single request can access an object with request scope.
Common Programming Error 10.6

One or both of the `<jsp:useBean>` attributes `class` and `type` must be specified; otherwise, a translation-time error occurs.

Many Web sites today place rotating advertisements on their Web pages. Each visit to one of these pages typically results in a different advertisement being displayed in the user’s Web browser. Typically, clicking an advertisement takes you to the Web site of the company that placed the advertisement. Our first example of `<jsp:useBean>` demonstrates a simple advertisement rotator bean that cycles through a list of five advertisements. In this example, the advertisements are covers for some of our books. Clicking a cover takes you to the Amazon.com Web site where you can read about and possibly order the book.

The `Rotator` bean (Fig. 10.17) has three methods: `getImage`, `getLink` and `nextAd`. Method `getImage` returns the image file name for the book cover image. Method `getLink` returns the hyperlink to the book at Amazon.com. Method `nextAd` updates the `Rotator` so the next calls to `getImage` and `getLink` return information for a different advertisement. Methods `getImage` and `getLink` each represent a read-only JavaBean property—`image` and `link`, respectively. `Rotator` keeps track of the current advertisement with its `selectedIndex` variable, which is updated by invoking method `nextAd`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The name used to manipulate the Java object with actions <code>&lt;jsp:setProperty&gt;</code> and <code>&lt;jsp:getProperty&gt;</code>. A variable of this name is also declared for use in JSP scripting elements. The name specified here is case sensitive.</td>
</tr>
<tr>
<td>scope</td>
<td>The scope in which the Java object is accessible—<code>page</code>, <code>request</code>, <code>session</code> or <code>application</code>. The default scope is <code>page</code>.</td>
</tr>
<tr>
<td>class</td>
<td>The fully qualified class name of the Java object.</td>
</tr>
<tr>
<td>beanName</td>
<td>The name of a bean that can be used with method <code>instantiate</code> of class <code>java.beans.Beans</code> to load a JavaBean into memory.</td>
</tr>
<tr>
<td>type</td>
<td>The type of the JavaBean. This can be the same type as the <code>class</code> attribute, a superclass of that type or an interface implemented by that type. The default value is the same as for attribute <code>class</code>. A <code>ClassCastException</code> occurs if the Java object is not of the type specified with attribute <code>type</code>.</td>
</tr>
</tbody>
</table>

Fig. 10.16 Attributes of the `<jsp:useBean>` action.

```
1 // Fig. 10.17: Rotator.java
2 // A JavaBean that rotates advertisements.
3 package com.deitel.advjhtp1.jsp.beans;
4
5 public class Rotator {
6    private String images[] = { "images/jhtp3.jpg",
7                               "images/xmlhtp1.jpg", "images/ebechtp1.jpg",
8                               "images/iw3htp1.jpg", "images/cpphtp3.jpg" };
9```

Fig. 10.17 `Rotator` bean that maintains a set of advertisements (part 1 of 2).
Chapter 10 JavaServer Pages (JSP)

Lines 7–8 of JavaServer Page adrotator.jsp (Fig. 10.18) obtain a reference to an instance of class Rotator. The id for the bean is rotator. The JSP uses this name to manipulate the bean. The scope of the object is session, so that each individual client will see the same sequence of ads during their browsing session. When adrotator.jsp receives a request from a new client, the JSP container creates the bean and stores it in JSP that client’s session (an HttpSession object). In each request to this JSP, line 22 uses the rotator reference created in line 7 to invoke the Rotator bean’s nextAd method. Thus, each request will receive the next advertisement maintained by the Rotator bean.

Lines 29–34 define a hyperlink to the Amazon.com site for a particular book. Lines 29–30 introduce action <jsp:getProperty> to obtain the value of the Rotator bean’s link property. Action <jsp:getProperty> has two attributes—name and property—that specify the bean object to manipulate and the property to get. If the JavaBean object uses standard JavaBean naming conventions, the method used to obtain the link property value from the bean should be getLink. Action <jsp:getProperty> invokes getLink on the bean referenced with rotator, converts the return value into a String and outputs the String as part of the response to the client. The link property becomes the value of the

```java
private String links[] = {
    "http://www.amazon.com/exec/obidos/ASIN/0130125075/" +
    "deitelassociatin",
    "http://www.amazon.com/exec/obidos/ASIN/0130284173/" +
    "deitelassociatin",
    "http://www.amazon.com/exec/obidos/ASIN/013028419X/" +
    "deitelassociatin",
    "http://www.amazon.com/exec/obidos/ASIN/0130161438/" +
    "deitelassociatin",
    "http://www.amazon.com/exec/obidos/ASIN/0130895717/" +
    "deitelassociatin" };
private int selectedIndex = 0;

// returns image file name for current ad
public String getImage()
{
    return images[selectedIndex];
}

// returns the URL for ad's corresponding Web site
public String getLink()
{
    return links[selectedIndex];
}

// update selectedIndex so next calls to getImage and
// getLink return a different advertisement
public void nextAd()
{
    selectedIndex = (selectedIndex + 1) % images.length;
}
```

Fig. 10.17 Rotator bean that maintains a set of advertisements (part 2 of 2).
hyperlink’s `href` attribute. The hyperlink is represented in the resulting Web page as the book cover image. Lines 32–33 create an `img` element and use another `<jsp:getProperty>` action to obtain the `Rotator` bean’s `image` property value.

Note that the link and image properties also can be obtained with JSP expressions. For example, the `<jsp:getProperty>` action in lines 29–30 can be replaced with the expression

```
<%= rotator.getLink() %>
```

Similarly, the `<jsp:getProperty>` action in lines 32–33 can be replaced with the expression

```
<%= rotator.getImage() %>
```

---

**Fig. 10.18** JSP `adrotator.jsp` uses a `Rotator` bean to display a different advertisement on each request to the page (part 1 of 2).
To test `adrotator.jsp` in Tomcat, copy `adrotator.jsp` into the `jsp` directory created in Section 10.3. You should have copied the `images` directory into the `jsp` directory when you tested Fig. 10.10. If not, you must copy the `images` directory there now. Copy `Rotator.class` into the `advjhtp1` Web application’s `WEB-INF\classes` directory in Tomcat. [Note: This example will work only if the proper package directory structure for `Rotator` is defined in the `classes` directory. `Rotator` is defined in package `com.deitel.advjhtp1.jsp.beans`.] Open your Web browser and enter the following URL to test `adrotator.jsp`:

```
http://localhost:8080/advjhtp1/jsp/adrotator.jsp
```

Try reloading this JSP several times in your browser to see the advertisement change with each request.

Action `<jsp:setProperty>` can set JavaBean property values. This action is particularly useful for mapping request parameter values to JavaBean properties. Request parameters can be used to set properties of primitive types `boolean`, `byte`, `char`, `int`, `long`, `float` and `double` and `java.lang` types `String`, `Boolean`, `Byte`, `Character`, `Integer`, `Long`, `Float` and `Double`. Figure 10.19 summarizes the `<jsp:setProperty>` attributes.
Common Programming Error 10.7

Use action `<jsp:setProperty>`’s value attribute to set JavaBean property types that cannot be set with request parameters; otherwise, conversion errors occur.

Software Engineering Observation 10.10

Action `<jsp:setProperty>` can use request-parameter values to set JavaBean properties only for properties of the following types: Strings, primitive types (boolean, byte, char, short, int, long, float and double) and type wrapper classes (Boolean, Byte, Character, Short, Integer, Long, Float and Double).

Our next example is a guest book that enables users to place their first name, last name and e-mail address into a guest book database. After submitting their information, users see a Web page containing all the users in the guest book. Each person’s e-mail address is displayed as a hyperlink that allows the user to send an e-mail message to the person. The example demonstrates action `<jsp:setProperty>`. In addition, the example introduces the JSP page directive and JSP error pages.

The guest book example consists of JavaBeans GuestBean (Fig. 10.20) and GuestDataBean (Fig. 10.21), and JSPs guestBookLogin.jsp (Fig. 10.22), guestBookView.jsp (Fig. 10.23) and guestBookErrorPage.jsp (Fig. 10.24). Sample outputs from this example are shown in Fig. 10.25.

JavaBean GuestBean (Fig. 10.20) defines three guest properties: firstName, lastName and email. Each is a read/write property with set and get methods to manipulate the property.
Chapter 10 JavaServer Pages (JSP) 623

JavaBean GuestDataBean (Fig. 10.21) connects to the guestbook database and provides methods getGuestList and addGuest to manipulate the database. The guestbook database has a single table (guests) containing three columns (firstName, lastName and email). We provide an SQL script (guestbook.sql) with this example that can be used with the Cloudscape DBMS to create the guestbook database. For further details on creating a database with Cloudscape, refer back to Chapter 8.
// Fig. 10.21: GuestDataBean.java
// Class GuestDataBean makes a database connection and supports
// inserting and retrieving data from the database.
package com.deitel.advjhtp1.jsp.beans;

// Java core packages
import java.io.*;
import java.sql.*;
import java.util.*;

public class GuestDataBean {
  private Connection connection;
  private PreparedStatement addRecord, getRecords;

  // construct TitlesBean object
  public GuestDataBean() throws Exception {
    // load the Cloudscape driver
    Class.forName( "COM.cloudscape.core.RmiJdbcDriver" );

    // connect to the database
    connection = DriverManager.getConnection( "jdbc:rmi:jdbc:cloudscape:guestbook" );

    getRecords =
        connection.prepareStatement(
            "SELECT firstName, lastName, email FROM guests"
        );

    addRecord =
        connection.prepareStatement(
            "INSERT INTO guests ( " +
                "firstName, lastName, email ) " +
            "VALUES ( ?, ?, ? )"
        );
  }

  // return an ArrayList of GuestBeans
  public ArrayList getGuestList() throws SQLException {
    ArrayList guestList = new ArrayList();

    // obtain list of titles
    ResultSet results = getRecords.executeQuery();

    // get row data
    while ( results.next() ) {
      GuestBean guest = new GuestBean();
      guest.setFirstName( results.getString( 1 ) );
      guest.setLastName( results.getString( 2 ) );
      guest.setEmail( results.getString( 3 ) );
    }
  }
}

Fig. 10.21 GuestDataBean performs database access on behalf of
guestBookLogin.jsp (part 1 of 2).
GuestDataBean \textit{method} \texttt{getGuestList} (lines 39–58) returns an \texttt{ArrayList} of \texttt{GuestBean} objects representing the guests in the database. Method \texttt{getGuestList} creates the \texttt{GuestBean} objects from the \texttt{ResultSet} returned by \texttt{PreparedStatement getRecords} (defined at lines 25–28 and executed at line 44).

\texttt{GuestDataBean} \textit{method} \texttt{addGuest} (lines 61–68) receives a \texttt{GuestBean} as an argument and uses the \texttt{GuestBean}'s properties as the arguments to \texttt{PreparedStatement addRecord} (defined at lines 30–35). This \texttt{PreparedStatement} (executed at line 67) inserts a new guest in the database.

Note that the \texttt{GuestDataBean}'s constructor, \texttt{getGuestList} and \texttt{addGuest} methods do not process potential exceptions. In the constructor, line 19 can throw a \texttt{ClassNotFoundException}, and the other statements can throw \texttt{SQLException}s. Similarly, \texttt{SQLException}s can be thrown from the bodies of methods \texttt{getGuestList} and \texttt{addGuest}. In this example, we purposely let any exceptions that occur get passed...
back to the JSP that invokes the GuestDataBean’s constructor or methods. This enables us to demonstrate JSP error pages. When a JSP performs an operation that causes an exception, the JSP can include scriptlets that catch the exception and process it. Exceptions that are not caught can be forwarded to a JSP error page for handling.

JavaServer Page guestBookLogin.jsp (Fig. 10.22) is a modified version of forward1.jsp (Fig. 10.11) that displays a form in which users can enter their first name, last name and e-mail address. When the user submits the form, guestBookLogin.jsp is requested again, so it can ensure that all the data values were entered. If not, the guestBookLogin.jsp responds with the form again, so the user can fill in missing field(s). If the user supplies all three pieces of information, guestBookLogin.jsp forwards the request to guestBookView.jsp, which displays the guest book contents.

---

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.22: guestBookLogin.jsp -->

<%@ page errorPage = "guestBookErrorPage.jsp" %>

<!-- beans used in this JSP -->
<jsp:useBean id = "guest" scope = "page"
   class = "com.deitel.advjhtp1.jsp.beans.GuestBean" />
<jsp:useBean id = "guestData" scope = "request"
   class = "com.deitel.advjhtp1.jsp.beans.GuestDataBean" />

<html xmlns = "http://www.w3.org/1999/xhtml">

<head>
   <title>Guest Book Login</title>

   <style type = "text/css">
      body {
         font-family: tahoma, helvetica, arial, sans-serif;
      }

      table, tr, td {
         font-size: .9em;
         border: 3px groove;
         padding: 5px;
         background-color: #dddddd;
      }
   </style>

</head>
```

---

Fig. 10.22 JavaServer page guestBookLogin.jsp enables the user to submit a first name, a last name and an e-mail address to be placed in the guest book (part 1 of 3).
Fig. 10.22 JavaServer page guestBookLogin.jsp enables the user to submit a first name, a last name and an e-mail address to be placed in the guest book (part 2 of 3).
Line 8 of guestBookLogin.jsp introduces the page directive, which defines information that is globally available in a JSP. Directives are delimited by `<%@` and `%>`. In this case, the page directive’s errorPage attribute is set to guestBookErrorPages.jsp (Fig. 10.24), indicating that all uncaught exceptions are forwarded to guestBookErrorPages.jsp for processing. A complete description of the page directive appears in Section 10.7.

Lines 11–14 define two `<jsp:useBean>` actions. Lines 11–12 create an instance of GuestBean called guest. This bean has page scope—it exists for use only in this page. Lines 14–14 create an instance of GuestDataBean called guestData. This bean has request scope—it exists for use in this page and any other page that helps process a single client request. Thus, when guestBookLogin.jsp forwards a request to guestBookView.jsp, the GuestDataBean is still available for use in guestBookView.jsp.

Line 36 demonstrates setting properties of the GuestBean called guest with request parameter values. The input elements on lines 55, 63 and 71 have the same names as the GuestBean properties. So, we use action `<jsp:setProperty>`’s ability to match request parameters to properties by specifying "*" for attribute property. Line 36 also can set the properties individually with the following lines:

```jsp
<jsp:setProperty name = "guest" property = "firstName"
    param = "firstName" />

<jsp:setProperty name = "guest" property = "lastName"
    param = "lastName" />

<jsp:setProperty name = "guest" property = "email"
    param = "email" />
```
If the request parameters had names that differed from GuestBean’s properties, the param attribute in each of the preceding <jsp:setProperty> actions could be changed to the appropriate request parameter name.

JavaServer Page guestBookView.jsp (Fig. 10.23) outputs an XHTML document containing the guest book entries in tabular format. Lines 8–10 define three page directives. Line 8 specifies that the error page for this JSP is guestBookErrorPage.jsp. Lines 9–10 introduce attribute import of the page directive. Attribute import enables programmers to specify Java classes and packages that are used in the context of the JSP. Line 9 indicates that classes from package java.util are used in this JSP, and line 10 indicates that classes from our package com.deitel.advjhtp1.jsp.beans also are used.

Lines 13–14 specify a <jsp:useBean> action that obtains a reference to a GuestDataBean object. If a GuestDataBean object already exists, the action returns a reference to the existing object. Otherwise, the action creates a GuestDataBean for use in this JSP. Lines 50–59 define a scriptlet that gets the guest list from the GuestDataBean and begin a loop to output the entries. Lines 61–70 combine fixed template text with JSP expressions to create rows in the table of guest book data that will be displayed on the client. The scriptlet at lines 72–76 terminates the loop.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.23: guestBookView.jsp -->

<%@ page errorPage = "guestBookErrorPage.jsp" %>
<%@ page import = "java.util.*" %>
<%@ page import = "com.deitel.advjhtp1.jsp.beans.*" %>

<%@-- GuestDataBean to obtain guest list --%>
<jsp:useBean id = "guestData" scope = "request"
class = "com.deitel.advjhtp1.jsp.beans.GuestDataBean" />

<html xmlns = "http://www.w3.org/1999/xhtml">

<head>
<title>Guest List</title>

<style type = "text/css">
body {
    font-family: tahoma, helvetica, arial, sans-serif;
}

table, tr, td, th {
    text-align: center;
    font-size: .9em;
} border: 3px groove;
padding: 5px;

</style>

</head>

```

Fig. 10.23 JavaServer page guestBookView.jsp displays the contents of the guest book (part 1 of 2).
Fig. 10.23 JavaServer page guestBookView.jsp displays the contents of the
guest book (part 2 of 2).
JavaServer Page guestBookErrorPage.jsp (Fig. 10.24) outputs an XHTML document containing an error message based on the type of exception that causes this error page to be invoked. Lines 8–10 define several page directives. Line 8 introduces page directive attribute isErrorPage. Setting this attribute to true makes the JSP an error page and enables access to the JSP implicit object exception that refers to an exception object indicating the problem that occurred.

Common Programming Error 10.8

JSP implicit object exception can be used only in error pages. Using this object in other JSPs results in a translation-time error.

Lines 29–46 define scriptlets that determine the type of exception that occurred and begin outputting an appropriate error message with fixed template data. The actual error message from the exception is output at line 56.
Figure 10.25 shows sample interactions between the user and the JSPs in the guest book example. In the first two rows of output, separate users entered their first name, last name and e-mail. In each case, the current contents of the guest book are returned and displayed for the user. In the final interaction, a third user specified an e-mail address that already existed in the database. The e-mail address is the primary key in the guests table of the guestbook database, so its values must be unique. Thus, the database prevents the new record from being inserted, and an exception occurs. The exception is forwarded to guestBookErrorPage.jsp for processing, which results in the last screen capture.

To test the guest book in Tomcat, copy guestBookLogin.jsp, guestBookView.jsp and guestBookErrorPage.jsp into the jsp directory created in Section 10.3. Copy GuestBean.class and GuestDataBean.class into the advjhtp1 Web application’s WEB-INF/classes directory in Tomcat. [Note: This example will work only if the proper package directory structure for GuestBean and GuestDataBean is defined in the classes directory. These classes are defined in package com.deitel.advjhtp1.jsp.beans.] Open your Web browser and enter the following URL to test guestBookLogin.jsp:

http://localhost:8080/advjhtp1/jsp/guestBookLogin.jsp
Fig. 10.25  JSP guest book sample output windows.
10.7 Directives

Directives are messages to the JSP container that enable the programmer to specify page settings (such as the error page), to include content from other resources and to specify custom-tag libraries for use in a JSP. Directives (delimited by `<%@` and `%>`) are processed at translation time. Thus, directives do not produce any immediate output, because they are processed before the JSP accepts any requests. Figure 10.26 summarizes the three directive types. These directives are discussed in the next several subsections.

10.7.1 page Directive

The page directive specifies global settings for the JSP in the JSP container. There can be many page directives, provided that there is only one occurrence of each attribute. The only exception to this rule is the import attribute, which can be used repeatedly to import Java packages used in the JSP. Figure 10.27 summarizes the attributes of the page directive.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>Defines page settings for the JSP container to process.</td>
</tr>
<tr>
<td>include</td>
<td>Causes the JSP container to perform a translation-time insertion of another resource’s content. As the JSP is translated into a servlet and compiled, the referenced file replaces the include directive and is translated as if it were originally part of the JSP.</td>
</tr>
<tr>
<td>taglib</td>
<td>Allows programmers to include their own new tags in the form of tag libraries. These libraries can be used to encapsulate functionality and simplify the coding of a JSP.</td>
</tr>
</tbody>
</table>

An SQLException occurred while interacting with the guestbook database.

The error message was:
The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint.

Please try again later.

Fig. 10.25 JSP guest book sample output windows.

Fig. 10.26 JSP directives.
Common Programming Error 10.9

Providing multiple page directives with one or more attributes in common is a JSP translation-time error.

Common Programming Error 10.10

Providing a page directive with an attribute or value that is not recognized is a JSP translation-time error.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>language</td>
<td>The scripting language used in the JSP. Currently, the only valid value for this attribute is java.</td>
</tr>
<tr>
<td>extends</td>
<td>Specifies the class from which the translated JSP will be inherited. This attribute must be a fully qualified package and class name.</td>
</tr>
<tr>
<td>import</td>
<td>Specifies a comma-separated list of fully qualified class names and/or packages that will be used in the current JSP. When the scripting language is java, the default import list is java.lang.<em>, javax.servlet.</em>, javax.servlet.jsp.<em>, and javax.servlet.http.</em>. If multiple import properties are specified, the package names are placed in a list by the container.</td>
</tr>
<tr>
<td>session</td>
<td>Specifies whether the page participates in a session. The values for this attribute are true (participates in a session—the default) or false (does not participate in a session). When the page is part of a session, the JSP implicit object session is available for use in the page. Otherwise, session is not available. In the latter case, using session in the scripting code results in a translation-time error.</td>
</tr>
<tr>
<td>buffer</td>
<td>Specifies the size of the output buffer used with the implicit object out. The value of this attribute can be none for no buffering, or a value such as 8kb (the default buffer size). The JSP specification indicates that the buffer used must be at least the size specified.</td>
</tr>
<tr>
<td>autoFlush</td>
<td>When set to true (the default value), this attribute indicates that the output buffer used with implicit object out should be flushed automatically when the buffer fills. If set to false, an exception occurs if the buffer overflows. This attribute’s value must be true if the buffer attribute is set to none.</td>
</tr>
<tr>
<td>isThreadSafe</td>
<td>Specifies if the page is thread safe. If true (the default), the page is considered to be thread safe, and it can process multiple requests at the same time. If false, the servlet that represents the page implements interface java.lang.SingleThreadModel and only one request can be processed by that JSP at a time. The JSP standard allows multiple instances of a JSP to exists for JSPs that are not thread safe. This enables the container to handle requests more efficiently. However, this does not guarantee that resources shared across JSP instances are accessed in a thread-safe manner.</td>
</tr>
<tr>
<td>info</td>
<td>Specifies an information string that describes the page. This string is returned by the getServletInfo method of the servlet that represents the translated JSP. This method can be invoked through the JSP’s implicit page object.</td>
</tr>
</tbody>
</table>
According to the JSP specification section 2.7.1, the \texttt{extends} attribute "should not be used without careful consideration as it restricts the ability of the JSP container to provide specialized superclasses that may improve on the quality of rendered service." Remember that a Java class can extend exactly one other class. If your JSP specifies an explicit superclass, the JSP container cannot translate your JSP into a subclass of one of the container application's own enhanced servlet classes.

\textbf{Common Programming Error 10.11}

Using JSP implicit object \texttt{session} in a JSP that does not have its \texttt{page} directive attribute \texttt{session} set to \texttt{true} is a translation-time error.

\section*{10.7.2 include Directive}

The \texttt{include} directive includes the content of another resource once, at JSP translation time. The \texttt{include} directive has only one attribute—\texttt{file}—that specifies the URL of the page to include. The difference between directive \texttt{include} and action \texttt{<jsp:include>} is noticeable only if the included content changes. For example, if the definition of an XHTML document changes after it is included with directive \texttt{include}, future invocations of the JSP will show the original content of the XHTML document, not the new content. In contrast, action \texttt{<jsp:include>} is processed in each request to the JSP. Therefore, changes to included content would be apparent in the next request to the JSP that uses action \texttt{<jsp:include>}.

\textbf{Software Engineering Observation 10.12}

The JavaServer Pages 1.1 specification does not provide a mechanism for updating text included in a JSP with the \texttt{include} directive. Version 1.2 of the JSP specification allows the container to provide such a mechanism, but the specification does not provide for this directly.

JavaServer Page \texttt{includeDirective.jsp} (Fig. 10.28) reimplements JavaServer Page \texttt{include.jsp} (Fig. 10.10) using \texttt{include} directives. To test \texttt{includeDirective.jsp} in Tomcat, copy \texttt{includeDirective.jsp} into the \texttt{jsp} directory created
in Section 10.3. Open your Web browser and enter the following URL to test include-Directive.jsp:

http://localhost:8080/advjhtp1/jsp/includeDirective.jsp

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.28: includeDirective.jsp -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Using the include directive</title>
<style type = "text/css">
body {
    font-family: tahoma, helvetica, arial, sans-serif;
}

    table, tr, td {
        font-size: .9em;
        border: 3px groove;
        padding: 5px;
        background-color: #dddddd;
    }
</style>
</head>
<body>
<table>
<tr>
<td style = "width: 160px; text-align: center">
<img src = "images/logotiny.png"
    width = "140" height = "93"
    alt = "Deitel & Associates, Inc. Logo" />
</td>
<td>

<!-- include banner.html in this JSP --%>
<%@ include file = "banner.html" %>

</td>
</tr>
<tr>
    <td>
     <!-- include toc.html in this JSP --%>
     <%@ include file = "toc.html" %>
    </td>
</tr>
</table>
</body>
</html>
```

**Fig. 10.28** JSP includeDirective.jsp demonstrates including content at translation-time with directive `include` (part 1 of 2).
10.8 Custom Tag Libraries

Throughout this chapter, you have seen how JavaServer Pages can simplify the delivery of dynamic Web content. Our discussion continues with JSP custom tag libraries, which provide another mechanism for encapsulating complex functionality for use in JSPs. Custom tag libraries define one or more custom tags that JSP implementors can use to create dynamic content. The functionality of these custom tags is defined in Java classes that implement interface `Tag` (package `javax.servlet.jsp.tagext`), normally by extending class `TagSupport` or `BodyTagSupport`. This mechanism enables Java programmers to create complex functionality for Web page designers who have no Java programming knowledge.
Previously, we introduced action `<jsp:useBean>` and JavaBeans to incorporate complex, encapsulated functionality in a JSP. In many cases, action `<jsp:useBean>` and JavaBeans can perform the same tasks as custom tags can. However, action `<jsp:useBean>` and JavaBeans have disadvantages—JavaBeans cannot manipulate JSP content and Web page designers must have some Java knowledge to use JavaBeans in a page. With custom tags, it is possible for Web page designers to use complex functionality without knowing any Java.

In this section, we present three examples of custom tags. Each tag is part of a single custom tag library that we refer to as `advjhtp1`. A JSP includes a custom tag library with the `taglib` directive. Figure 10.29 summarizes the `taglib` directive’s attributes.

Each of the examples in this section uses directive `taglib`. There are several types of custom tags that have different levels of complexity. We demonstrate simple tags, simple tags with attributes and tags that can process their body elements. For complete details on custom tag libraries, see the resources in Section 10.9.

### 10.8.1 Simple Custom Tag

Our first custom tag example implements a simple custom tag that inserts the string “Welcome to JSP Tag Libraries” in a JSP. When implementing custom tags, you must define a tag-handler class for each tag that implements the tag’s functionality, a tag library descriptor that provides information about the tag library and its custom tags to the JSP container and a JSP that uses the custom tag. Figure 10.30 (`customTagWelcome.jsp`) demonstrates our first custom tag. At the end of this section, we discuss how to configure this example for testing on Tomcat.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uri</code></td>
<td>Specifies the relative or absolute URI of the tag library descriptor.</td>
</tr>
<tr>
<td><code>tagPrefix</code></td>
<td>Specifies the required prefix that distinguishes custom tags from built-in tags. The prefix names <code>jsp</code>, <code>jspx</code>, <code>java</code>, <code>javax</code>, <code>servlet</code>, <code>sun</code> and <code>sunw</code> are reserved.</td>
</tr>
</tbody>
</table>

**Fig. 10.29** Attributes of the `taglib` directive.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>

<!-- Fig. 10.30: customTagWelcome.jsp -->
<!-- JSP that uses a custom tag to output content. -->

<!-- taglib directive --%->
<%@ taglib uri = "advjhtp1-taglib.tld" prefix = "advjhtp1" %>

<html xmlns = "http://www.w3.org/1999/xhtml">
```

**Fig. 10.30** JSP `customTagWelcome.jsp` uses a simple custom tag (part 1 of 2).
The `taglib` directive at line 9 enables the JSP to use the tags in our tag library. The directive specifies the `uri` of the tag library descriptor file (`advjhtp1-taglib.tld`; Fig. 10.32) that provides information about our tag library to the JSP container and the `prefix` for each tag (`advjhtp1`). JSP programmers use the tag library `prefix` when referring to tags in a specific tag library. Line 20 uses a custom tag called `welcome` to insert text in the JSP. Note that the prefix `advjhtp1:` precedes the tag name. This enables the JSP container to interpret the meaning of the tag and invoke the appropriate `tag handler`. Also note that line 20 can be written with start and end tags as follows:

```html
<advjhtp1:welcome> </advjhtp1:welcome>
```

Figure 10.31 defines class `WelcomeTagHandler`—the tag handler that implements the functionality of our custom tag `welcome`. Every tag handler must implement interface `Tag`, which defines the methods a JSP container invokes to incorporate a tag’s functionality in a JSP. Most tag handler classes implement interface `Tag` by extending either class `TagSupport` or class `BodyTagSupport`.

Software Engineering Observation 10.13

Classes that define custom tag handlers must implement interface `Tag` from package `javax.servlet.jsp.tagext`.

Software Engineering Observation 10.14

A custom tag handler class should extend class `TagSupport` if the body of the tag is ignored or simply output during custom tag processing.
Software Engineering Observation 10.15
A custom tag handler class should extend class `BodyTagSupport` if the handler interacts with the tag’s body content.

Software Engineering Observation 10.16
Custom tag handlers must be defined in Java packages.

Class `WelcomeTagHandler` implements interface `Tag` by extending class `TagSupport` (both from package `javax.servlet.jsp.tagext`). The most important methods of interface `Tag` are `doStartTag` and `doEndTag`. The JSP container invokes these methods when it encounters the starting custom tag and the ending custom tag, respectively. These methods throw `JspException` if problems are encountered during custom-tag processing.

```java
// Fig. 10.31: WelcomeTagHandler.java
// Custom tag handler that handles a simple tag.
package com.deitel.advjhtp1.jsp.taglibrary;

// Java core packages
import java.io.*;

// Java extension packages
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class WelcomeTagHandler extends TagSupport {

    // Method called to begin tag processing
    public int doStartTag() throws JspException {
        // attempt tag processing
        try {
            // obtain JspWriter to output content
            JspWriter out = pageContext.getOut();

            // output content
            out.print("Welcome to JSP Tag Libraries!");
        }

        // rethrow IOException to JSP container as JspException
        catch( IOException ioException ) {
            throw new JspException( ioException.getMessage() );
        }

        return SKIP_BODY; // ignore the tag's body
    }

    // Fig. 10.31 WelcomeTagHandler custom tag handler.
```

Software Engineering Observation 10.17

If exceptions other than JspExceptions occur in a custom tag handler class, the exceptions should be caught and processed. If such exceptions would prevent proper tag processing, the exceptions should be rethrown as JspExceptions.

In this example, class WelcomeTagHandler overrides method doStartTag to output text that becomes part of the JSP’s response. Line 20 uses the custom tag handler’s pageContext object (inherited from class TagSupport) to obtain the JSP’s JspWriter object that method doStartTag uses to output text. Line 23 uses the JspWriter to output a string. Line 31 returns the static integer constant SKIP_BODY (defined in interface Tag) to indicate that the JSP container should ignore any text or other elements that appear in the tag’s body. To include the body content as part of the response, specify static integer constant EVAL_BODY_INCLUDE as the return value. This example does not require any processing when the ending tag is encountered by the JSP container, so we did not override doEndTag.

Figure 10.32 defines the custom tag library descriptor file. This XML document specifies information required by the JSP container such as the version number of the tag library (element tlibversion), the JSP version number (element jspversion), information about the library (element info) and information about the tags in the library (one tag element for each tag). In this tag library descriptor, the tag element at lines 18–30 describes our welcome tag. Line 19 specifies the tag’s name—used by JSP programmers to access the custom functionality in a JSP. Lines 21–23 specify the tagclass—the custom tag handler class. This element associates the tag name with a specific tag handler class. Element bodycontent (line 25) specifies that our custom tag has an empty body. This value can also be tagdependent or JSP. Lines 27–29 specify information about the tag with an info element. [Note: We introduce other elements of the tag library descriptor as necessary. For a complete description of the tag library descriptor, see the JavaServer Pages 1.1 specification, which can be downloaded from java.sun.com/products/jsp/download.html.]

Software Engineering Observation 10.18

The custom tag handler class must be specified with its full package name in the tagclass element of the tag library descriptor.

```xml
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<!-- a tag library descriptor -->
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>advjhtpl</shortname>
</taglib>
```

Fig. 10.32 Custom tag library descriptor file advjhtpl-taglib.tld (part 1 of 2).
To test customTagWelcome.jsp in Tomcat, copy customTagWelcome.jsp and advjhtpl-taglib.tld into the jsp directory created in Section 10.3. Copy WelcomeTagHandler.class into the advjhtpl Web application’s WEB-INF\classes directory in Tomcat. [Note: Class WelcomeTagHandler must appear in its proper package director structure in classes directory. WelcomeTagHandler is defined in package com.deitel.advjhtpl.jsp.taglibrary.] Open your Web browser and enter the following URL to test customTagWelcome.jsp:

http://localhost:8080/advjhtpl/jsp/customTagWelcome.jsp

10.8.2 Custom Tag with Attributes

Many XHTML and JSP elements use attributes to customize functionality. For example, an XHTML element can specify a style attribute that indicates how the element should be formatted in a client’s Web browser. Similarly, the JSP action elements have attributes that help customize their behavior in a JSP. Our next example demonstrates how to specify attributes for your custom tags.

Figure 10.33 (customTagAttribute.jsp) is similar to Fig. 10.30. This example uses a new tag called, welcome2, to insert text in the JSP that is customized based on the value of attribute firstName. The screen capture shows the results of the welcome2 tags on lines 20 and 30. The tag at line 20 specifies the value "Paul" for attribute firstName. Lines 26–28 define a scriptlet that obtains the value of request parameter name and assign it to String reference name. Line 30 uses the name in a JSP expression as the value for the firstName attribute. In the sample screen capture, this JSP was invoked with the following URL:

http://localhost:8080/advjhtpl/jsp/customTagAttribute.jsp?firstName=Sean
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 10.x: customTagAttribute.jsp             -->
<!-- JSP that uses a custom tag to output content. -->

<%@ taglib uri = "advjhtpl-taglib.tld" prefix = "advjhtpl" %>

<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Specifying Custom Tag Attributes</title>
</head>
<body>
<p>Demonstrating an attribute with a string value</p>
<h1>
<advjhtpl:welcome2 firstName = "Paul" />
</h1>

<p>Demonstrating an attribute with an expression value</p>
<h1>
<!-- scriptlet to obtain "name" request parameter -->
<% String name = request.getParameter( "name" ); %>
<advjhtpl:welcome2 firstName = "<%= name %>">
</h1>
</body>
</html>
When defining the custom tag handler for a tag with attributes, you must provide methods that enable the JSP container to set the attribute values in the tag handler. Methods that manipulate attributes follow the same set- and get-method naming conventions as do JavaBean properties. Thus, the custom tag’s firstName attribute is set with method setFirstName. Similarly, the method to obtain the firstName attribute’s value would be getFirstName (we did not define this method for this example). Class Welcome2TagHandler (Fig. 10.34) defines its firstName variable at line 13 and a corresponding set method setFirstName (lines 37–40). When the JSP container encounters a welcome2 tag in a JSP, it creates a new Welcome2TagHandler object to process the tag and sets the tag’s attributes. Next, the container invokes method doStartTag (lines 16–34) to perform the custom tag processing. Lines 24–25 use the firstName attribute value as part of the text output by the custom tag.

```java
1 // Fig. 10.34: Welcome2TagHandler.java
2 // Custom tag handler that handles a simple tag.
3 package com.deitel.advjhtp1.jsp.taglibrary;
4
5 // Java core packages
6 import java.io.*;
7
8 // Java extension packages
9 import javax.servlet.jsp.*;
10 import javax.servlet.jsp.tagext./*;
11
12 public class Welcome2TagHandler extends TagSupport {
13     private String firstName = "";
14
15     // Method called to begin tag processing
16     public int doStartTag() throws JspException
17     {
18         // attempt tag processing
19         try {
20             // obtain JspWriter to output content
21             JspWriter out = pageContext.getOut();
22
23             // output content
24             out.print( "Hello " + firstName +
25                     ", <br />Welcome to JSP Tag Libraries!" );
26         }
27         // rethrow IOException to JSP container as JspException
28         catch( IOException ioException ) {
29             throw new JspException( ioException.getMessage() );
30         }
31
32         return SKIP_BODY; // ignore the tag's body
33     }
34 }
```

Fig. 10.34 Welcome2TagHandler custom tag handler for a tag with an attribute (part 1 of 2).
Before the welcome2 tag can be used in a JSP, we must make the JSP container aware of the tag by adding it to a tag library. To do this, add the tag element of Fig. 10.35 as a child of element taglib in the tag library descriptor advjhtp1-taglib.tld. As in the previous example, element tag contains elements name, tagclass, bodycontent and info. Lines 16–20 introduce element attribute for specifying the characteristics of a tag’s attributes. Each attribute must have a separate attribute element that contains the name, required and rtexprvalue elements. Element name (line 17) specifies the attribute’s name. Element required specifies whether the attribute is required (true) or optional (false). Element rtexprvalue specifies whether the value of the attribute can be the result of a JSP expression evaluated at runtime (true) or whether it must be a string literal (false).

To test customTagAttribute.jsp in Tomcat, copy customTagAttribute.jsp and the updated advjhtp1-taglib.tld into the jsp directory created in Section 10.3. Copy Welcome2TagHandler.class into the advjhtp1 Web application’s WEB-INF\classes directory in Tomcat. [Note: This example will work only if the proper package-directory structure for Welcome2TagHandler is defined in the classes directory.] Open your Web browser and enter the following URL to test customTagAttribute.jsp:

http://localhost:8080/advjhtp1/jsp/customTagAttribute.jsp?firstName=Sean

The text ?firstName=Sean in the preceding URL specifies the value for request parameter name that is used by the custom tag welcome2 at line 30 in Fig. 10.33.

```java
// set firstName attribute to the users first name
public void setFirstName( String username )
{
    firstName = username;
}
```

Fig. 10.34 Welcome2TagHandler custom tag handler for a tag with an attribute (part 2 of 2).

```xml
<!-- A tag with an attribute -->
<tag>
  <name>welcome2</name>
  <tagclass>
    com.deitel.advjhtp1.jsp.taglibrary.Welcome2TagHandler
  </tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    Inserts content welcoming user to tag libraries. Uses attribute "name" to insert the user's name.
  </info>
</tag>
```

Fig. 10.35 Element tag for the welcome2 custom tag (part 1 of 2).
10.8.3 Evaluating the Body of a Custom Tag

Custom tags are particularly powerful for processing the element body. When a custom tag interacts with the element body, additional methods are required to perform those interactions. The methods are defined in class \texttt{BodyTagSupport}. In our next example, we re-implement \texttt{guestBookView.jsp} (Fig. 10.23) and replace the JavaBean processing performed in the JSP with a custom \texttt{guestlist} tag.

Figure 10.36 (\texttt{customTagBody.jsp}) uses the custom \texttt{guestlist} tag at lines 41–52. Note that the JSP expressions in the body of element \texttt{guestlist} use variable names that are not defined in the JSP. These variables are defined by the custom tag handler when the custom tag is encountered. The custom tag handler places the variables in the JSP’s \texttt{PageContext}, so the variables can be used throughout the page. Although no repetition is defined in the JSP, the custom tag handler is defined to iterate over all the guests in the \texttt{guestbook} database. This action results in the creation of a table row in the resulting Web page for each guest in the database.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Guest List</title>
<style type = "text/css">
body {
  font-family: tahoma, helvetica, arial, sans-serif
}

table, tr, td, th {
  text-align: center;
  font-size: .9em;
  border: 3px groove;
</style>
</head>
</html>
```
As in guestBookView.jsp, the custom tag handler GuestBookTag (Fig. 10.37) creates a GuestDataBean to access the guestbook database. Class GuestBookTag extends BodyTagSupport, which contains several new methods including doInit-
**Body** and **doAfterBody** (from interface **BodyTag**). Method **doInitBody** is called once, after **doStartTag** and before **doAfterBody**. Method **doAfterBody** can be called many times to process the body of a custom tag.

**Software Engineering Observation 10.19**

Method **doInitBody** typically performs one-time processing before method **doAfterBody** processes the body of a custom tag. If method **doStartTag** returns **Tag.SKIP_BODY**, method **doInitBody** will not be called.

```java
// Fig. 10.37: GuestBookTag.java
// Custom tag handler that reads information from the guestbook
// database and makes that data available in a JSP.
package com.deitel.advjhtp1.jsp.taglibrary;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Deitel packages
import com.deitel.advjhtp1.jsp.beans.*;

public class GuestBookTag extends BodyTagSupport {
    private String firstName;
    private String lastName;
    private String email;

    private GuestDataBean guestData;
    private GuestBean guest;
    private Iterator iterator;

    // Method called to begin tag processing
    public int doStartTag() throws JspException {
        try {
            guestData = new GuestDataBean();
            List list = guestData.getGuestList();
            iterator = list.iterator();
            if (iterator.hasNext()) {
                processNextGuest();
                return EVAL_BODY_TAG;  // continue body processing
            } else
                return SKIP_BODY;        // terminate body processing
        }
    }
}
```

**Fig. 10.37** GuestBookTag custom tag handler (part 1 of 2).
44  // if any exceptions occur, do not continue processing
45  // tag's body
46  catch( Exception exception ) {
47      exception.printStackTrace();
48      return SKIP_BODY;  // ignore the tag's body
49  }
50  }
51  }
52  // process body and determine if body processing
53  // should continue
54  public int doAfterBody()
55  {
56      // attempt to output body data
57      try {
58          bodyContent.writeOut( getPreviousOut() );
59      }
60      }
61      // if exception occurs, terminate body processing
62      catch ( IOException ioException ) {
63          ioException.printStackTrace();
64          return SKIP_BODY;        // terminate body processing
65      }
66      bodyContent.clearBody();
67      if ( iterator.hasNext() ) {
68          processNextGuest();
69          return EVAL_BODY_TAG;  // continue body processing
70      } else
71          return SKIP_BODY;        // terminate body processing
72  }
73  // obtains the next GuestBean and extracts its data
74  private void processNextGuest()
75  {
76      // get next guest
77      guest = ( GuestBean ) iterator.next();
78      pageContext.setAttribute(  
79          "firstName", guest.getFirstName() );
80      pageContext.setAttribute(  
81          "lastName", guest.getLastName() );
82      pageContext.setAttribute(  
83          "email", guest.getEmail() );
84  }
85  }
The JSP container invokes method `doStartTag` (lines 27–51) when it encounters the custom `guestlist` tag in a JSP. Lines 31–34 create a new `GuestDataBean`, obtain a `List` of `GuestBeans` from the `GuestDataBean` and create an `Iterator` for manipulating the `ArrayList` contents. If there are no elements in the list (tested at line 36), line 42 returns `SKIP_BODY` to indicate that the container should perform no further processing of the `guestlist` tag’s body. Otherwise, line 37 invokes private method `processNextGuest` (lines 80–93) to extract the information for the first guest and create variables containing that information in the JSP’s `PageContext` (represented with variable `pageContext` that was inherited from `BodyTagSupport`). Method `processNextGuest` uses `PageContext` method `setAttribute` to specify each variable’s name and value. The container is responsible for creating the actual variables used in the JSP. This is accomplished with the help of class `GuestBookTagExtraInfo` (Fig. 10.38).

Method `doAfterBody` (lines 55–77) performs the repetitive processing of the `guestlist` tag’s body. The JSP container determines whether method `doAfterBody` should be called again, based on the method’s return value. If `doAfterBody` returns `EVAL_BODY_TAG`, the container calls method `doAfterBody` again. If `doAfterBody` returns `SKIP_BODY`, the container stops processing the body and invokes the custom tag handler’s `doEndTag` method to complete the custom processing. Line 59 invokes `writeOut` on variable `bodyContent` (inherited from `BodyTagSupport`) to process the first client’s data (stored when `doStartTag` was called). Variable `bodyContent` refers to an object of class `BodyContent` from package `javax.servlet.jsp.tagext`. The argument to method `writeOut` is the result of method `getPreviousOut` (inherited from class `BodyTagSupport`), which returns the `JspWriter` object for the JSP that invokes the custom tag. This enables the custom tag to continue building the response to the client using the same output stream as the JSP. Next, line 68 invokes `bodyContent`’s method `clearBody` to ensure that the body content that was just output does not get processed as part of the next call to `doAfterBody`. Lines 70–76 determine whether there are more guests to process. If so, `doAfterBody` invokes private method `processNextGuest` to obtain the data for the next guest and returns `EVAL_BODY_TAG` to indicate that the container should call `doAfterBody` again. Otherwise, `doAfterBody` returns `SKIP_BODY` to terminate processing of the body.

The JSP container cannot create variables in the `PageContext` unless the container knows the names and types of those variables. This information is specified by a class with the same name as the custom tag handler and that ends with `ExtraInfo` (`GuestBookTagExtraInfo` in Fig. 10.38). `ExtraInfo` classes extend class `TagExtraInfo` (package `javax.servlet.jsp.tagext`). The container uses the information specified by a subclass of `TagExtraInfo` to determine what variables it should create (or use) in the `PageContext`. To specify variable information, override method `getVariableInfo`. This method returns an array of `VariableInfo` objects that the container uses either to create new variables in the `PageContext` or to enable a custom tag to use existing variables in the `PageContext`. The `VariableInfo` constructor receives four arguments—a `String` representing the name of the variable, a `String` representing the variable’s class name, a `boolean` indicating whether or not the variable should be created by the container (`true` if so) and a `static` integer constant representing the variable’s scope in the JSP. The constants in class `VariableInfo` are `NESTED`, `AT_BEGIN` and
AT_END. NESTED indicates that the variable can be used only in the custom tag’s body. AT_BEGIN indicates that the variable can be used anywhere in the JSP after the starting tag of the custom tag is encountered. AT_END indicates that the variable can be used anywhere in the JSP after the ending tag of the custom tag.

Before the guestlist tag can be used in a JSP, we must make the JSP container aware of the tag by adding it to a tag library. To do this, add the tag element of Fig. 10.39 as a child of element taglib in the tag library descriptor advjhtpl-taglib.tld. As in the previous example, element tag contains elements name, tagclass, bodycontent and info. Lines 10–12 introduce element teiclass for specifying the custom tag’s ExtraInfo class.

To test customTagBody.jsp in Tomcat, copy customTagBody.jsp and the updated advjhtpl-taglib.tld into the jsp directory created in Section 10.3. Copy GuestBookTag and GuestBookTagExtraInfo.class into the advjhtpl Web application’s WEB-INF\classes directory in Tomcat. [Note: This example will work only if the proper package directory structure for GuestBookTag and GuestBookTagExtraInfo is defined in the classes directory.] Open your Web browser and enter the following URL to test customTagBody.jsp:

http://localhost:8080/advjhtpl/jsp/customTagBody.jsp

Fig. 10.38 GuestBookTagExtraInfo used by the container to define scripting variables in a JSP that uses the guestlist custom tag.
This chapter has presented many JSP capabilities. However, there are additional features that are beyond the scope of this book. For a complete description of JavaServer Pages, see the JavaServer Pages 1.1 specification, which can be downloaded from java.sun.com/products/jsp/download.html. Other JSP resources are listed in Section 10.9. The next chapter continues our JSP and servlet discussion by presenting a substantial e-business case study in which we build an online bookstore. The case study integrates many topics discussed up to this point in the text, including JDBC, servlets, JSP and XML. Also, we discuss several additional servlet features as part of the case study. The techniques shown in the case study provide a foundation for the capstone e-business case study presented in Chapters 16 through 19.

10.9 World Wide Web Resources

java.sun.com/products/jsp
The home page for information about JavaServer Pages at the Sun Microsystems Java site.

java.sun.com/products/servlet
The home page for information about servlets at the Sun Microsystems Java site.

java.sun.com/j2ee
The home page for the Java 2 Enterprise Edition at the Sun Microsystems Java site.

www.w3.org
The World Wide Web Consortium home page. This site provides information about current and developing Internet and Web standards, such as XHTML, XML and CSS.

jsptags.com
This site includes tutorials, tag libraries, software and other resources for JSP programmers.

jspinsider.com
This Web programming site concentrates on resources for JSP programmers. It includes software, tutorials, articles, sample code, references and links to other JSP and Web programming resources.
SUMMARY

- JavaServer Pages (JSPs) are an extension of servlet technology.
- JavaServer Pages enable Web application programmers to create dynamic content by reusing pre-defined components and by interacting with components using server-side scripting.
- JSP programmers can create custom tag libraries that enable Web-page designers who are not familiar with Java programming to enhance their Web pages with powerful dynamic content and processing capabilities.
- Classes and interfaces specific to JavaServer Pages programming are located in packages `javax.servlet.jsp` and `javax.servlet.jsp.tagext`.
- The JavaServer Pages 1.1 specification can be downloaded from `java.sun.com/products/jsp/download.html`.
- There are four key components to JSPs—directives, actions, scriptlets and tag libraries.
- Directives specify global information that is not associated with a particular JSP request.
- Actions encapsulate functionality in predefined tags that programmers can embed in a JSP.
- Scriptlets, or scripting elements, enable programmers to insert Java code that interacts with components in a JSP (and possibly other Web application components) to perform request processing.
- Tag libraries are part of the tag extension mechanism that enables programmers to create new tags that encapsulate complex Java functionality.
- JSPs normally include XHTML or XML markup. Such markup is known as fixed template data or fixed template text.
- Programmers tend to use JSPs when most of the content sent to the client is fixed template data and only a small portion of the content is generated dynamically with Java code.
- Programmers use servlets when a small portion of the content is fixed template data.
- JSPs normally execute as part of a Web server. The server often is referred to as the JSP container.
- When a JSP-enabled server receives the first request for a JSP, the JSP container translates that JSP into a Java servlet that handles the current request and future requests to the JSP.
- The JSP container places the Java statements that implement a JSP’s response in method `_jspService` at translation time.
- The request/response mechanism and life cycle of a JSP are the same as those of a servlet.
- JSPs can define methods `jspInit` and `jspDestroy` that are invoked when the container initializes a JSP and when the container terminates a JSP, respectively.
- JSP expressions are delimited by `<%=` and `%>`. Such expressions are converted to Strings by the JSP container and are output as part of the response.
- The XHTML `meta` element can set a refresh interval for a document that is loaded into a browser. This causes the browser to request the document repeatedly at the specified interval in seconds.
- When you first invoke a JSP in Tomcat, there is a delay as Tomcat translates the JSP into a servlet and invokes the servlet to respond to your request.
- Implicit objects provide programmers with servlet capabilities in the context of a JavaServer Page.
- Implicit objects have four scopes—application, page, request and session.
- Objects with application scope are part of the JSP and servlet container application.
- Objects with page scope exist only as part of the page in which they are used. Each page has its own instances of the page-scope implicit objects.
- Objects with request scope exist for the duration of the request. Request-scope objects go out of scope when request processing completes with a response to the client.
• Objects with session scope exist for the client’s entire browsing session.
• JSP scripting components include scriptlets, comments, expressions, declarations and escape sequences.
• Scriptlets are blocks of code delimited by `<%` and `%>`. They contain Java statements that are placed in method `_jspService` when the container translates a JSP into a servlet.
• JSP comments are delimited by `<--` and `-->`. XHTML comments are delimited by `<!--` and `-->`. Java’s single-line comments (`//`) and multiline comments (delimited by `/*` and `*/`) can be used inside scriptlets.
• JSP comments and scripting language comments are ignored and do not appear in the response.
• A JSP expression, delimited by `<%=` and `%>`, contains a Java expression that is evaluated when a client requests the JSP containing the expression. The container converts the result of a JSP expression to a `String` object, then outputs the `String` as part of the response to the client.
• Declarations, delimited by `<%!` and `%>`, enable a JSP programmer to define variables and methods. Variables become instance variables of the class that represents the translated JSP. Similarly, methods become members of the class that represents the translated JSP.
• Special characters or character sequences that the JSP container normally uses to delimit JSP code can be included in a JSP as literal characters in scripting elements, fixed template data and attribute values by using escape sequences.
• JSP standard actions provide JSP implementors with access to several of the most common tasks performed in a JSP. JSP containers process actions at request time.
• JavaServer Pages support two include mechanisms—the `<jsp:include>` action and the `include` directive.
• Action `<jsp:include>` enables dynamic content to be included in a JavaServer Page. If the included resource changes between requests, the next request to the JSP containing the `<jsp:include>` action includes the new content of the resource.
• The `include` directive is processed once, at JSP translation time, and causes the content to be copied into the JSP. If the included resource changes, the new content will not be reflected in the JSP that used the include directive unless that JSP is recompiled.
• Action `<jsp:forward>` enables a JSP to forward the processing of a request to a different resource. Processing of the request by the original JSP terminates as soon as the request is forwarded.
• Action `<jsp:param>` specifies name/value pairs of information that are passed to the `include`, `forward` and `plugin` actions. Every `<jsp:param>` action has two required attributes—`name` and `value`. If a `param` action specifies a parameter that already exists in the request, the new value for the parameter takes precedence over the original value. All values for that parameter can be obtained with the JSP implicit object `request`’s `getParameterValues` method, which returns an array of `Strings`.
• JSP action `<jsp:plugin>` enables an applet or JavaBean to be added to a Web page in the form of a browser-specific `object` or `embed` XHTML element. This action also enables the downloading and installation of the Java Plug-in if it is not already installed on the client computer.
• Action `<jsp:useBean>` enables a JSP to manipulate a Java object. This action can be used to create a Java object for use in the JSP or to locate an existing object.
• Like JSP implicit objects, objects specified with action `<jsp:useBean>` have page, request, session or application scope that indicates where they can be used in a Web application.
• Action `<jsp:getProperty>` obtains the value of JavaBean’s property. Action `<jsp:getProperty>` has two attributes—`name` and `property`—that specify the bean object to manipulate and the property to get.
• JavaBean property values can be set with action `<jsp:setProperty>`. This action is particularly useful for mapping request parameter values to JavaBean properties. Request parameters can be used to set properties of primitive types `boolean`, `byte`, `char`, `int`, `long`, `float` and `double` and `java.lang` types `String`, `Boolean`, `Byte`, `Character`, `Integer`, `Long`, `Float` and `Double`.

• The `page` directive defines information that is globally available in a JSP. Directives are delimited by `<%@` and `%>`. The `page` directive’s `errorPage` attribute indicates where all uncaught exceptions are forwarded for processing.

• Action `<jsp:setProperty>` has the ability to match request parameters to properties of the same name in a bean by specifying "*" for attribute `property`.

• Attribute `import` of the `page` directive enables programmers to specify Java classes and packages that are used in the context of a JSP.

• If the attribute `isErrorPage` of the `page` directive is set to `true`, the JSP is an error page. This condition enables access to the JSP implicit object `exception` that refers to an exception object indicating the problem that occurred.

• Directives are messages to the JSP container that enable the programmer to specify page settings (such as the error page), to include content from other resources and to specify custom tag libraries that can be used in a JSP. Directives are processed at the time a JSP is translated into a servlet and compiled. Thus, directives do not produce any immediate output.

• The `page` directive specifies global settings for a JSP in the JSP container. There can be many `page` directives, provided that there is only one occurrence of each attribute. The exception to this rule is the `import` attribute, which can be used repeatedly to import Java packages.

• Custom tag libraries define one or more custom tags that JSP implementors can use to create dynamic content. The functionality of these custom tags is defined in Java classes that implement interface `Tag` (package `javax.servlet.jsp.tagext`), normally by extending class `TagSupport` or `BodyTagSupport`.

• A JSP can include a custom tag library with the `taglib` directive.

• When implementing custom tags, you must define a tag handler class for each tag that provides the tag’s functionality, a tag library descriptor that provides information about the tag library and its custom tags to the JSP container and a JSP that uses the custom tag.

• The most important methods of interface `Tag` are `doStartTag` and `doEndTag`. The JSP container invokes these methods when it encounters the starting custom tag and the ending custom tag, respectively.

• A custom tag library descriptor file is an XML document that specifies information about the tag library that is required by the JSP container.

• Class `BodyTagSupport` contains several methods for interacting with the body of a custom tag, including `doInitBody` and `doAfterBody` (from interface `BodyTag`). Method `doInitBody` is called once after `doStartTag` and once before `doAfterBody`. Method `doAfterBody` can be called many times to process the body of a custom tag.

**TERMINOLOGY**

`%` escape sequence for `%`
`<! -->` XHTML comment delimiters
`<%--` and `-->` JSP comment delimiters
`<%` and `%>` scriptlet delimiters
`<%!` and `%>` declaration delimiters
`<%=` and `%>` JSP expression delimiters
`
`<%@` and `%>` directive delimiters
`
`<% escape sequence for <%
`action`
`align` attribute of `<jsp:plugin>` action
`application` implicit object
`application scope`
SELF-REVIEW EXERCISES

10.1 Fill in the blanks in each of the following statements:
   a) JSP action ________ enables an applet or JavaBean to be added to a Web page in the
      form of a browser-specific object or embed XHTML element.
   b) Action ________ has the ability to match request parameters to properties of the same
      name in a bean by specifying "*" for attribute property.
   c) There are four key components to JSPs: ________, ________, ________, and ________.
   d) A JSP can include a custom tag library with the ________ directive.
   e) The implicit objects have four scopes: ________, ________, ________, and ________.
   f) The ________ directive is processed once, at JSP translation time and causes content
      to be copied into the JSP.
   g) Classes and interfaces specific to JavaServer Pages programming are located in packages
      ________, ________, ________, and ________.
   h) JSPs normally execute as part of a Web server that is referred to as the ________.
   i) Method ________ can be called repeatedly to process the body of a custom tag.
   j) JSP scripting components include ________, ________, ________, ________, ________, and ________.

10.2 State whether each of the following is true or false. If false, explain why.
   a) An object with page scope exists in every JSP of a particular Web application.
   b) Directives specify global information that is not associated with a particular JSP request.
   c) The JSP container invokes methods doInitBody and doAfterBody when it encounters
      the starting custom tag and the ending custom tag, respectively.
   d) Tag libraries are part of the tag extension mechanism that enables programmers to create
      new tags that encapsulate complex Java functionality.
   e) Action <jsp:include> is evaluated once at page translation time.
f) Like XHTML comments, JSP comments and script language comments appear in the response to the client.

g) Objects with application scope are part of a particular Web application.
h) Each page has its own instances of the page-scope implicit objects.
i) Action `<jsp:setProperty>` has the ability to match request parameters to properties of the same name in a bean by specifying `"*"` for attribute `property`.

j) Objects with session scope exist for the client’s entire browsing session.

**ANSWERS TO SELF-REVIEW EXERCISES**

10.1  
   a) `<jsp:plugin>`  
   b) `<jsp:setProperty>`  
   c) directives, actions, scriptlets, tag libraries  
   d) `taglib`  
   e) application, page, request and session  
   f) `include`  
   g) `javax.servlet.jsp`, `javax.servlet.jsp.tagext`  
   h) JSP container  
   i) `doAfterBody`  
   j) scriptlets, comments, expressions, declarations, escape sequences.

10.2  
   a) False. Objects with page scope exist only as part of the page in which they are used.  
   b) True.  
   c) False. The JSP container invokes methods `doStartTag` and `doEndTag` when it encounters the starting custom tag and the ending custom tag, respectively.  
   d) True.  
   e) False. Action `<jsp:include>` enables dynamic content to be included in a JavaServer Page.  
   f) False. JSP comments and script language comments are ignored and do not appear in the response.  
   g) False. Objects with application scope are part of the JSP and servlet container application.  
   h) True.  
   i) True.  
   j) True.

**EXERCISES**

10.3  
   Create class `ResultSetTag` (a custom tag handler) that can display information from any `ResultSet`. Use class `GuestBookTag` of Fig. 10.37 as a guide. The `pageContext` attribute names should be the column names in the `ResultSet`. The column names can be obtained through the `ResultSetMetaData` associated with the `ResultSet`. Create the tag library descriptor for the custom tag in this exercise and test the custom tag in a JSP.

10.4  
   Create a JSP and JDBC-based address book. Use the guest book example of Fig. 10.20 through Fig. 10.24 as a guide. Your address book should allow one to insert entries, delete entries and search for entries.

10.5  
   Incorporate the `ResultSetTag` of Exercise 10.3 into the address book example in Exercise 10.4.

10.6  
   Reimplement your solution to Exercise 9.5 (Dynamic Web FAQs) using JSPs rather than servlets. Create a custom tag handler similar to the one you created in Exercise 10.3 to help display the FAQs information.

10.7  
   Modify your solution to Exercise 10.6 so that the first JSP invoked by the user returns a list of FAQs topics from which to choose. Each topic should be a hyperlink that invokes another JSP with an argument indicating which topic the user would like to view. The JSP should query the FAQs database and return an XHTML document containing only FAQs for that topic.

10.8  
   Reimplement the Web application of Fig. 9.27 (favorite animal survey) using JSPs.

10.9  
   Modify your solution to Exercise 10.8 to allow the user to see the survey results without responding to the survey.

10.10  
   Reimplement Fig. 9.24 (book recommendations) using JSPs. Use the JSP implicit object `session` to track the user’s selections and determine appropriate book recommendations. Remember to use the `page` directive to indicate that each JSP participates in a session.
Case Study: Servlet and JSP Bookstore

Objectives

• To build a three-tier, client/server, distributed Web application using Java servlet and JavaServer Pages technology.
• To be able to perform servlet/JSP interactions.
• To be able to use a RequestDispatcher to forward requests to another resource for further processing.
• To be able to create XML from a servlet and XSL transformations to convert the XML into a format the client can display.
• To introduce the Java 2 Enterprise Edition reference implementation server.
• To be able to deploy a Web application using the Java 2 Enterprise Edition.

The world is a book, and those who do not travel, read only a page.
Saint Augustine

If we do not lay out ourselves in the service of mankind, whom should we serve?
John Adams

We must take the current when it serves, or lose our ventures.
William Shakespeare
Chapter 11 Case Study: Servlet and JSP Bookstore

11.1 Introduction

This chapter serves as a capstone for our presentation of JSP and servlets. Here, we implement a bookstore Web application that integrates JDBC, XML, JSP and servlet technologies. The case study introduces additional servlet features that are discussed as they are encountered in the case study.

This chapter also serves as an introduction to the *Java 2 Enterprise Edition 1.2.1 reference implementation* used in Chapters 14–18. Unlike the JSP and servlet chapters, which demonstrated examples using Apache’s Tomcat JSP and servlet container, this chapter deploys the bookstore application on the J2EE 1.2.1 reference implementation application server software, which is downloadable from [java.sun.com/j2ee/download.html](http://java.sun.com/j2ee/download.html) (see Appendix E for installation and configuration instructions). The J2EE 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container. After reading this chapter, you will be able to implement a substantial distributed Web application with many components, and you will be able to deploy that application on the J2EE 1.2.1 application server.

Outline

| 11.1 Introduction |
| 11.2 Bookstore Architecture |
| 11.3 Entering the Bookstore |
| 11.4 Obtaining the Book List from the Database |
| 11.5 Viewing a Book’s Details |
| 11.6 Adding an Item to the Shopping Cart |
| 11.7 Viewing the Shopping Cart |
| 11.8 Checking Out |
| 11.9 Processing the Order |
| 11.10 Deploying the Bookstore Application in J2EE 1.2.1 |
| 11.10.1 Configuring the *books* Data Source |
| 11.10.2 Launching the Cloudscape Database and J2EE Servers |
| 11.10.3 Launching the J2EE Application Deployment Tool |
| 11.10.4 Creating the Bookstore Application |
| 11.10.5 Creating *BookServlet* and *AddToCartServlet* Web Components |
| 11.10.6 Adding Non-Servlet Components to the Application |
| 11.10.7 Specifying the Web Context, Resource References, JNDI Names and Welcome Files |
| 11.10.8 Deploying and Executing the Application |

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises
11.2 Bookstore Architecture

This section overviews the architecture of the Bug2Bug.com bookstore application. We present a diagram of the basic interactions between XHTML documents, JSPs and servlets. Also, we present a table of all the documents and classes used in the case study. Our sample outputs demonstrate how the XHTML documents sent to the client are rendered.

Our Bug2Bug.com shopping-cart case study consists of a series of XHTML documents, JSPs and servlets that interact to simulate a bookstore selling Deitel publications. This case study is implemented as a distributed, three-tier, Web-based application. The client tier is represented by the user’s Web browser. The browser displays static XHTML documents and dynamically created XHTML documents that allow the user to interact with the server tier. The server tier consists of several JSPs and servlets that act on behalf of the client. These JSPs and servlets perform tasks such as creating a list of publications, creating documents containing the details about a publication, adding items to the shopping cart, viewing the shopping cart and processing the final order. Some of the JSPs and servlets perform database interactions on behalf of the client.

The database tier uses the books database introduced in Chapter 8, Java Database Connectivity. In this case study, we use only the titles table from the database (see Chapter 8).

Figure 11.1 illustrates the interactions between the bookstore’s application components. In the diagram, names without file extensions (displayBook and addToCart) represent servlet aliases (i.e., the names used to invoke the servlets). As you will see when we deploy the case study in Section 11.10, the Java 2 Enterprise Edition 1.2.1 reference implementation includes an Application Deployment Tool. Among its many features, this tool enables us to specify the alias used to invoke a servlet. For example, addToCart is the alias for servlet AddToCartServlet. The Application Deployment Tool creates the deployment descriptor for a servlet as part of deploying an application.

After the application is deployed, users can visit the bookstore by entering the following URL in a browser:

http://localhost:8000/advjhtpl/store/
This URL requests the default home page for the store (**index.html**). The user can view the list of products by clicking a button on the home page. This invokes **books.jsp**, which interacts with a database to create the list of books dynamically. The result is an XHTML document containing links to the servlet with alias **displayBook**. This servlet receives as a parameter the ISBN number for the selected book and returns an XHTML document containing the information for that book. From this document, the user can click buttons to place the current book in the shopping cart or view the shopping cart. Adding a book to a shopping cart invokes the servlet with alias **addToCart**. Viewing the contents of the cart invokes **viewCart.jsp** to return an XHTML document containing the contents of the cart, subtotals the dollar cost of each item and a total dollar cost of all the items in the cart. When the user adds an item to the shopping cart, the **addToCart** servlet processes the user’s request, then forwards it to **viewCart.jsp** to create the document that displays the current cart. At this point, the user can either continue shopping (**books.jsp**) or proceed to checkout (**order.html**). In the latter case, the user is presented with a form to input name, address and credit-card information. Then, the user submits the form to invoke **process.jsp**, which completes the transaction by sending a confirmation document to the user.

Figure 11.2 overviews the XHTML documents, JSPs, servlets, JavaBeans and other files used in this case study.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>index.html</strong></td>
<td>This is the default home page for the bookstore, which is displayed by entering the following URL in the client’s Web browser:</td>
</tr>
<tr>
<td></td>
<td><strong><a href="http://localhost:8000/advjhtp1/store">http://localhost:8000/advjhtp1/store</a></strong></td>
</tr>
<tr>
<td><strong>styles.css</strong></td>
<td>This Cascading Style Sheet (CSS) file is linked to all XHTML documents rendered on the client. The CSS file allows us to apply uniform formatting across all the XHTML static and dynamic documents rendered.</td>
</tr>
<tr>
<td><strong>books.jsp</strong></td>
<td>This JSP uses <strong>BookBean</strong> objects and a <strong>TitlesBean</strong> object to create an XHTML document containing the product list.</td>
</tr>
<tr>
<td></td>
<td>The <strong>TitlesBean</strong> object queries the <strong>books</strong> database to obtain the list of titles in the database. The results are processed and placed in an ArrayList of <strong>BookBean</strong> objects.</td>
</tr>
<tr>
<td></td>
<td>The list is stored as a session attribute for the client.</td>
</tr>
<tr>
<td><strong>BookBean.java</strong></td>
<td>An instance of this JavaBean represents the data for one book.</td>
</tr>
<tr>
<td></td>
<td>The bean’s <strong>getXML</strong> method returns an XML <strong>Element</strong> representing the book.</td>
</tr>
<tr>
<td><strong>TitlesBean.java</strong></td>
<td>JSP <strong>books.jsp</strong> uses an instance of this JavaBean to obtain an ArrayList containing a <strong>BookBean</strong> for every product in the database.</td>
</tr>
</tbody>
</table>
11.3 Entering the Bookstore

Figure 11.3 (index.html) is the default home page for the Bug2Bug.com bookstore. This file is also known as the welcome file—an option specified at application deployment time (see Section 11.10). When the bookstore application is running on your computer in the Java 2 Enterprise Edition 1.2.1 reference implementation, you can enter the following URL in your Web browser to display the home page:

http://localhost:8000/advjhttp1/store
Lines 11–12 specify a linked style sheet **styles.css** (Fig. 11.4). All XHTML documents sent to the client use this style sheet, so that uniform formatting can be applied to
the documents. The form at lines 24–27 provides a submit button that enables you to enter the store. Clicking this button invokes books.jsp (Fig. 11.7), which creates and returns an XHTML document containing the product list.

Figure 11.4 (styles.css) defines the common styles for rendering XHTML documents in this case study. Lines 1–2 indicate that all text in the body element should be centered and that the background color of the body should be steel blue. The background color is represented by the hexadecimal number #b0c4de. Line 3 defines class .bold to apply bold font weight to text. Lines 4–7 define class .bigFont with four CSS attributes. Elements to which this class is applied appear in bold, Helvetica font, which is double the size of the base-text font. The color of the font is dark blue (represented by the hexadecimal number #00008b). If the Helvetica font is not available, the browser will attempt to use Arial, and then the generic font sans-serif as a last resort. Line 8 defines class .italic to apply italic font style to text. Line 9 defines class .right to right justify text. Lines 10–11 indicate that all table, th (table head data) and td (table data) elements should have a three-pixel, grooved border with five pixels of internal padding between the text in a table cell and the border of that cell. Lines 12–14 indicate that all table elements should have a bright-blue background color (represented by the hexadecimal number #6495ed) and that all table elements should use automatically determined margins on both their left and right sides. This causes the table to be centered on the page. Not all of these styles are used in every XHTML document. However, using a single linked style sheet allows us to change the look and feel of our store quickly and easily by modifying the CSS file. For more information on CSS visit

www.w3.org/Style/CSS

At this Web site, you will find the CSS specifications. Each specification includes an index of all the current CSS attributes and their permitted values.

Portability Tip 11.1

Different browsers have different levels of support for Cascading Style Sheets.

```
1  body          { text-align: center;  
2                   background-color: #B0C4DE; }  
3  .bold         { font-weight: bold; }  
4  .bigFont      { font-family: helvetica, arial, sans-serif;  
5                   font-weight: bold;  
6                   font-size: 2em;  
7                   color: #00008B; }  
8  .italic       { font-style: italic; }  
9  .right        { text-align: right; }  
10   table, th, td { border: 3px groove;  
11                 padding: 5px; }  
12   table        { background-color: #6495ed;  
13                 margin-left: auto;  
14                 margin-right: auto  
```

Fig. 11.4  Shared cascading style sheet (styles.css) used to apply common formatting across XHTML documents rendered on the client.
11.4 Obtaining the Book List from the Database

JavaServer Pages often generate XHTML that is sent to the client for rendering. JSP books.jsp (Fig. 11.7) generates an XHTML document containing a list of hyperlinks to information about each book in the titles table of the books database. From this list, the user can view information about a particular book by clicking the hyperlink for that book. This JSP uses a TitlesBean (Fig. 11.5) object and BookBean (Fig. 11.6) objects to create the product list. Each of the JavaBeans and books.jsp are discussed in this section. Figure 11.7 shows the rendering of the XHTML document sent to the browser by books.jsp.

The TitlesBean (Fig. 11.5) JavaBean performs a database query to obtain the list of titles in the database. Then, the results are processed and placed in an ArrayList of BookBean objects. As we will see in Fig. 11.7, the ArrayList is stored by books.jsp as a session attribute for the client.

```
1 // TitlesBean.java
2 // Class TitlesBean makes a database connection and retrieves
3 // the books from the database.
4 package com.deitel.advjhtp1.store;
5
6 // Java core packages
7 import java.io.*;
8 import java.sql.*;
9 import java.util.*;
10
11 // Java extension packages
12 import javax.naming.*;
13 import javax.sql.*;
14
15 public class TitlesBean implements Serializable {
16     private Connection connection;
17     private PreparedStatement titlesQuery;
18
19     //construct TitlesBean object
20     public TitlesBean() {
21         // attempt database connection and setup SQL statements
22         try {
23             InitialContext ic = new InitialContext();
24             DataSource source =
25                 (DataSource) ic.lookup(
26                     "java:comp/env/jdbc/books");
27             connection = source.getConnection();
28         }
29     }
30 }
```

Fig. 11.5 TitlesBean for obtaining book information from the books database and creating an ArrayList of BookBean objects (part 1 of 3).
titlesQuery =
    connection.prepareStatement(
        "SELECT isbn, title, editionNumber, " +
        "copyright, publisherID, imageFile, price " +
        "FROM titles ORDER BY title"
    );
}

// process exceptions during database setup
catch ( SQLException sqlException ) {
    sqlException.printStackTrace();
}

// process problems locating data source
catch ( NamingException namingException ) {
    namingException.printStackTrace();
}

// return a List of BookBeans
public List getTitles() {
    List titlesList = new ArrayList();

    // obtain list of titles
    try {
        ResultSet results = titlesQuery.executeQuery();

        // get row data
        while ( results.next() ) {
            BookBean book = new BookBean();

            book.setISBN( results.getString( "isbn" ) );
            book.setTitle( results.getString( "title" ) );
            book.setEditionNumber( results.getInt( "editionNumber" ) );
            book.setCopyright( results.getString( "copyright" ) );
            book.setPublisherID( results.getInt( "publisherID" ) );
            book.setImageFile( results.getString( "imageFile" ) );
            book.setPrice( results.getDouble( "price" ) );

            titlesList.add( book );
        }
    }

    // process exceptions during database query
    catch ( SQLException exception ) {
        exception.printStackTrace();
    }
}

Fig. 11.5 TitlesBean for obtaining book information from the books database and creating an ArrayList of BookBean objects (part 2 of 3).
The JavaBean TitlesBean requires us to introduce the Java Naming and Directory Interface (JNDI). Enterprise Java applications often access information and resources (such as databases) that are external to those applications. In some cases, those resources are distributed across a network. Just as an RMI client uses the RMI registry to locate a server object so the client can request a server, Enterprise application components must be able to locate the resources they use. An Enterprise Java application container must provide a naming service that implements JNDI and enables the components executing in that container to perform name lookups to locate resources. The J2EE 1.2.1 reference implementation server includes such a naming service that we use to locate our books database at execution time.

The TitlesBean uses JNDI to interact with the naming service and locate the data source (i.e., the books database). The TitlesBean constructor (lines 20–49) attempts the connection to the database using class InitialContext from package javax.naming and interface DataSource from package javax.sql. [These packages must be available to your compiler to compile this example.] When you deploy an Enterprise Java application (see Section 11.10), you specify the resources (such as databases) used by the application and the JNDI names for those resources. Using an InitialContext, an Enterprise application component can look up a resource. The InitialContext provides access to the application’s naming environment.

Line 24 creates a new InitialContext. The InitialContext constructor throws a NamingException if it cannot locate a naming service. Lines 26–28 invoke InitialContext method lookup to locate our books data source. In the argument, the text java:comp/env indicates that method lookup should search for the resource in the application’s component environment entries (i.e., the resource names specified at deployment time). The text jdbc/books indicates that the resource is a JDBC data source called books. Method lookup returns a DataSource object and throws a

```java
83     // return the list of titles
84     finally {
85         return titlesList;
86     }
87     }
88
89     // close statements and terminate database connection
90     protected void finalize()
91     {
92         // attempt to close database connection
93         try {
94             connection.close();
95         }
96
97         // process SQLException on close operation
98         catch ( SQLException sqlException ) {
99             sqlException.printStackTrace();
100         }
101     }
102 }
```

Fig. 11.5 TitlesBean for obtaining book information from the books database and creating an ArrayList of BookBean objects (part 3 of 3).
NamingException if it cannot resolve the name it receives as an argument. Line 25 uses the DataSource to connect to the database. Lines 32–37 create a PreparedStatement that, when executed, returns the information about each title from the titles table of the books database.

Method getTitles (lines 52–87) returns a List (titlesList) containing a BookBean for each title in the database. Line 58 executes titlesQuery. Lines 57–76 process the ResultSet (results). For each row in results, line 62 creates a new BookBean, and lines 64–72 set the attributes of the BookBean to columns in that ResultSet row. ResultSet methods getString, getInt and getDouble return the column data in the appropriate formats. Line 74 adds the new BookBean to titlesList. In the finally block, titlesList is returned. If there is an exception while performing the database interactions or if there are no records in the database, the List will be empty.

An instance of the BookBean (Fig. 11.6) JavaBean represents the properties for one book, including the book’s ISBN number, title, copyright, cover image file name, edition number, publisher ID number and price. Each of these properties is a read/write property. Some of this information is not used in this example. BookBean method getXML returns an XML Element representing the book.

```java
package com.deitel.advjhtp1.store;

public class BookBean implements Serializable {

    private String ISBN, title, copyright, imageFile;
    private int editionNumber, publisherID;
    private double price;

    // set ISBN number
    public void setISBN( String isbn )
    {
        ISBN = isbn;
    }

    // return ISBN number
    public String getISBN()
    {
        return ISBN;
    }

    // set title
    public void setTitle( String title )
    {
        this.title = title;
    }

    // return title
    public String getTitle()
    {
        return title;
    }

    // set copyright
    public void setCopyright( String copyright )
    {
        this.copyright = copyright;
    }

    // return copyright
    public String getCopyright()
    {
        return copyright;
    }

    // set edition number
    public void setEditionNumber( int editionNumber )
    {
        this.editionNumber = editionNumber;
    }

    // return edition number
    public int getEditionNumber()
    {
        return editionNumber;
    }

    // set publisher ID number
    public void setPublisherID( int publisherID )
    {
        this.publisherID = publisherID;
    }

    // return publisher ID number
    public int getPublisherID()
    {
        return publisherID;
    }

    // set price
    public void setPrice( double price )
    {
        this.price = price;
    }

    // return price
    public double getPrice()
    {
        return price;
    }

    // set image file name
    public void setImageFile( String imageFile )
    {
        this.imageFile = imageFile;
    }

    // return image file name
    public String getImageFile()
    {
        return imageFile;
    }

    public String getXML()
    {
        String xml = 
```
// set book title
public void setTitle( String bookTitle )
{
    title = bookTitle;
}

// return book title
public String getTitle()
{
    return title;
}

// set copyright year
public void setCopyright( String year )
{
    copyright = year;
}

// return copyright year
public String getCopyright()
{
    return copyright;
}

// set file name of image representing product cover
public void setImageFile( String fileName )
{
    imageFile = fileName;
}

// return file name of image representing product cover
public String getImageFile()
{
    return imageFile;
}

// set edition number
public void setEditionNumber( int edition )
{
    editionNumber = edition;
}

// return edition number
public int getEditionNumber()
{
    return editionNumber;
}

// set publisher ID number
public void setPublisherID( int id )
{
publisherID = id;
}

// return publisher ID number
public int getPublisherID()
{
    return publisherID;
}

// set price
public void setPrice( double amount )
{
    price = amount;
}

// return price
public double getPrice()
{
    return price;
}

// get an XML representation of the Product
public Element getXML( Document document )
{
    // create product root element
    Element product = document.createElement( "product" );

    // create isbn element, append as child of product
    Element temp = document.createElement( "isbn" );
    temp.appendChild( document.createTextNode( getISBN() ) );
    product.appendChild( temp );

    // create title element, append as child of product
    temp = document.createElement( "title" );
    temp.appendChild( document.createTextNode( getTitle() ) );
    product.appendChild( temp );

    // create a currency formatting object for US dollars
    NumberFormat priceFormatter =
        NumberFormat.getCurrencyInstance( Locale.US );

    // create price element, append as child of product
    temp = document.createElement( "price" );
    temp.appendChild( document.createTextNode(
        priceFormatter.format( getPrice() ) ) );
    product.appendChild( temp );

    // create imageFile element, append as child of product
    temp = document.createElement( "imageFile" );
    temp.appendChild( document.createTextNode( getImageFile() ) );
    product.appendChild( temp );

Fig. 11.6 BookBean that represents a single book’s information and defines the XML format of that information (part 3 of 4).
Method `getXML` (lines 103–154) uses the `org.w3c.dom` package’s `Document` and `Element` interfaces to create an XML representation of the book data as part of the `Document` that is passed to the method as an argument. The complete information for one book is placed in a `product` element (created at line 106). The elements for the individual properties of a book are appended to the `product` element as children. For example, line 109 uses `Document` method `createElement` to create element `isbn`. Line 110 uses `Document` method `createTextNode` to specify the text in the `isbn` element, and `Element` method `appendChild` to append the text to element `isbn`. Then, line 111 appends element `isbn` as a child of element `product` with `Element` method `appendChild`. Similar operations are performed for the other book properties. Lines 119–120 obtain a `NumberFormat` object that formats currency for the U. S. locale to format the book price in US dollars (line 125). Line 150 returns element `product` to the caller. We revisit method `getXML` in our `BookServlet` discussion (Fig. 11.8). For more information about XML and Java, refer to Appendices A–F.

JavaServer Page `books.jsp` dynamically generates the list of titles as an XHTML document to be rendered on the client. Lines 7–11 specify the JSP page settings. This JSP uses classes from our store package (`com.deitel.advjhtp1.store`) and package `java.util`. Also, this JSP uses session-tracking features. The dynamic parts of this JSP are defined in lines 31–64 with JSP scriptlets and expressions.
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- books.jsp -->

<%@ page language = "java" import = "com.deitel.advjhtp1.store.*, java.util.*" session = "true" %>

<!-- begin document -->

<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Book List</title>
<link rel = "stylesheet" href = "styles.css" type = "text/css" />
</head>
<body>
<p class = "bigFont">Available Books</p>
<p class = "bold">Click a link to view book information</p>
<p>
<TitlesBean titlesBean = new TitlesBean();
List titles = titlesBean.getTitles();
BookBean currentBook;

// store titles in session for further use
session.setAttribute( "titles", titles );

// obtain an Iterator to the set of keys in the List
Iterator iterator = titles.iterator();

// use the Iterator to get each BookBean and create
// a link to each book
while ( iterator.hasNext() ) {
    currentBook = ( BookBean ) iterator.next();

%> <%-- end scriptlet to insert literal XHTML and --%>
%-- JSP expressions output from this loop --%>
%-- link to a book's information --%>
<span class = "bold">

Fig. 11.7  JSP books.jsp returns to the client an XHTML document containing the book list (part 1 of 2).
The scriptlet begins at line 31. Line 32 creates a `TitlesBean`, and line 33 invokes its `getTitles` method to obtain the `List` of `BookBean` objects. Line 37 sets a `titles` session attribute to store the `List` for use later in the client’s session. Line 40 obtains an

`<a href = "displayBook?isbn=<%= currentBook.getISBN() %>">
<%= currentBook.getTitle() + ", " +
currentBook.getEditionNumber() + "e" %>
</a>
</span><br />

// continue scriptlet

} // end while loop

<!-- end scriptlet -->

</p>
</body>
</html>
Iterator for the List. Lines 44–45 begin a loop that uses the Iterator to output each hyperlink. The scriptlet temporarily terminates here so that lines 51–58 can insert XHTML markup. In this markup, line 53 uses a JSP expression to insert the book’s ISBN number as the value in a name/value pair that is passed to the displayBook servlet (BookServlet) as an argument. Lines 55–56 use another JSP expression to insert the book’s title and edition number as the text displayed for the hyperlink. Lines 60–64 continue the scriptlet with the closing curly brace of the while loop that started at line 44.

11.5 Viewing a Book’s Details

Like many companies, Bug2Bug.com is beginning to use XML on its Web site. When the user selects a book in books.jsp, Bug2Bug.com converts the book’s information to XML. BookServlet (Fig. 11.8) transforms the XML representation of the book into an XHTML document using XSL style sheet book.xsl (Fig. 11.9).

There are two major parts in BookServlet’s doGet method (lines 24–103). Lines 28–62 locate the BookBean for the book selected by the user in books.jsp. Lines 65–102 process the XML representation of a book and apply an XSL transformation to that XML.

```java
// BookServlet.java
// Servlet to return one book's information to the client.
// The servlet produces XML which is transformed with XSL to
// produce the client XHTML page.
package com.deitel.advjhtp1.store;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// third-party packages
import org.w3c.dom.*;
import org.xml.sax.*;

public class BookServlet extends HttpServlet {
    protected void doGet( HttpServletRequest request,
        HttpServletResponse response )
        throws ServletException, IOException
    {
        HttpSession session = request.getSession( false );
```

Fig. 11.8 BookServlet obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client (part 1 of 4).
// RequestDispatcher to forward client to bookstore home page if no session exists or no books are selected
RequestDispatcher dispatcher = request.getRequestDispatcher( "index.html" );

// if session does not exist, forward to index.html
if ( session == null )
    dispatcher.forward( request, response );

// get books from session object
List titles =
    ( List ) session.getAttribute( "titles" );

// locate BookBean object for selected book
Iterator iterator = titles.iterator();
BookBean book = null;

String isbn = request.getParameter( "isbn" );

while ( iterator.hasNext() ) {
    book = ( BookBean ) iterator.next();

    if ( isbn.equals( book.getISBN() ) ) {
        // save the book in a session attribute
        session.setAttribute( "bookToAdd", book );
        break; // isbn matches current book
    }
}

// if book is not in list, forward to index.html
if ( book == null )
    dispatcher.forward( request, response );

// get XML document and transform for browser client
try {
    // get a DocumentBuilderFactory for creating
    // a DocumentBuilder (i.e., an XML parser)
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

    // get a DocumentBuilder for building the DOM tree
    DocumentBuilder builder =
        factory.newDocumentBuilder();

    // create a new Document (empty DOM tree)
    Document messageDocument = builder.newDocument();

    // get XML from BookBean and append to Document
    Element bookElement = book.getXML( messageDocument );
    messageDocument.appendChild( bookElement );
Fig. 11.8 BookServlet obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client (part 3 of 4).
Line 28 obtains the HttpSession object for the current client. This object contains a session attribute indicating the book selected by the user in books.jsp. Lines 32–33 obtain a RequestDispatcher for the "index.html" document by calling ServletRequest method getRequestDispatcher. A RequestDispatcher (package javax.servlet) provides two methods—forward and include—that enable a servlet to forward a client request to another resource or include content from another resource in a servlet’s response. In this example, if there is no session object for the current client (lines 36–37) or if there is no book selected (lines 61–62), the request is forwarded back to the index.html home page of our bookstore. Methods forward and include each take two arguments—the HttpServletRequest and HttpServletResponse objects for the current request.

Note that RequestDispatcher objects can be obtained with method getRequestDispatcher from an object that implements interface ServletRequest or from the ServletContext with methods getRequestDispatcher or getNamedDispatcher. ServletContext method getNamedDispatcher receives the name of a servlet as an argument, then searches the ServletContext for a servlet by that name. If no such servlet is found, the method returns null. Both the ServletRequest and the ServletContext getRequestDispatcher methods simply return to the client browser the content of the specified path if the path does not represent a servlet.

Lines 40–41 get the List of BookBeans from the session object. Lines 44–58 perform a linear search to locate the BookBean for the selected book. (Note: For larger databases, it would be more appropriate to use a Map rather than a List.) The ISBN number for that book is stored in an isbn parameter passed to the servlet (retrieved at line 47). If the BookBean is found, line 55 sets session attribute bookToAdd with that BookBean as the attribute’s value. AddToCartServlet (Fig. 11.10) uses this attribute to update the shopping cart.

The try block (lines 65–97) performs the XML and XSL processing that result in an XHTML document containing a single book’s information. Before the XML and XSL capabilities can be used, you must download and install Sun’s Java API for XML Parsing (JAXP) version 1.1 from java.sun.com/xml/download.htm. The root directory of JAXP (jaxp-1.1) contains three JAR files—crimson.jar, jaxp.jar and xalan.jar—that are required for compiling and running programs that use JAXP. These files must be added to the Java extension mechanism for your Java 2 Standard Edi-
tion installation. Place a copy of these files in your Java installation’s extensions directory (\texttt{jre/lib/ext} on Linux/UNIX, and \texttt{jre\lib\ext} on Windows).

Software Engineering Observation 11.1

\textit{JAXP 1.1 is part of the J2EE 1.3 reference implementation.}

Creating a Document Object Model (DOM) tree from an XML document requires a \texttt{DocumentBuilder} parser object. \texttt{DocumentBuilder} objects are obtained from a \texttt{DocumentBuilderFactory}. Lines 68–69 obtain a \texttt{DocumentBuilderFactory}. Lines 72–73 obtain a \texttt{DocumentBuilder} parser object that enables the program to create a \texttt{Document} object tree in which the XML document elements are represented as \texttt{Element} objects. Line 76 uses the \texttt{DocumentBuilder} object to create a new \texttt{Document}. Line 79 invokes the \texttt{BookBean}'s \texttt{getXML} method to obtain an \texttt{Element} representation of the book. Line 80 appends this \texttt{Element} to \texttt{messageDocument} (the \texttt{Document} object). Classes \texttt{DocumentBuilderFactory} and \texttt{DocumentBuilder} are located in package \texttt{javax.xml.parsers}. Classes \texttt{Document} and \texttt{Element} are located in package \texttt{org.w3c.dom}. [\textit{Note: For detailed information on XML, see Appendices A–D.}]

Next, line 83 specifies the response content type, and line 84 obtains a \texttt{PrintWriter} to output the response to the client. Lines 87–89 create an \texttt{InputStream} that will be used by the XSL transformation processor to read the XSL file. The response is created by the XSL transformation performed in method \texttt{transform} (lines 107–137). We pass three arguments to this method—the XML \texttt{Document} to which the XSL transformation will be applied (\texttt{messageDocument}), the \texttt{InputStream} that reads the XSL file (\texttt{xslStream}) and the target stream to which the results should be written (\texttt{out}). The output target can be one of several types, including a character stream (i.e., the \texttt{response} object’s \texttt{PrintWriter} in this example).

Line 112 creates a \texttt{DOMSource} that represents the XML document. This serves as the source of the XSL to transform. Lines 115–116 create a \texttt{StreamSource} for the XSL file. This serves as the source of the XSL that transforms the \texttt{DOMSource}. Line 119 creates a \texttt{StreamResult} for the \texttt{PrintWriter} to which the results of the XSL transformation are written. Lines 122–123 create a \texttt{TransformerFactory} with \texttt{static} method \texttt{newInstance}. This object enables the program to obtain a \texttt{Transformer} object that applies the XSL transformation. Lines 126–127 create a \texttt{Transformer} using \texttt{TransformerFactory} method \texttt{newTransformer}, which receives a \texttt{StreamSource} argument representing the XSL (\texttt{xslSource} in this example). Line 130 invokes \texttt{Transformer} method \texttt{transform} to perform the XSL transformation on the given \texttt{DOMSource} object (\texttt{xmlSource}) and writes the result to the given \texttt{StreamResult} object (\texttt{result}). Lines 134–136 catch a \texttt{TransformerException} if a problem occurs when creating the \texttt{TransformerFactory}, creating the \texttt{Transformer} or performing the transformation.

Figure 11.9 contains the \texttt{book.xsl} style sheet file used in the XSL transformation. The values of six elements in the XML document are placed in the resulting XHTML document. Lines 23 and 30 place the book’s \texttt{title} in the document’s \texttt{title} element and in a paragraph at the beginning of the document’s \texttt{body} element, respectively. Line 36 specifies an \texttt{img} element in which the value of the \texttt{imageFile} element of an XML
The document specifies the name of the file representing the book’s cover image. Line 37 specifies the `alt` attribute of the `img` element using the book’s `title`. Lines 43, 51, 59 and 67 place the book’s `price`, `isbn`, `editionNumber` and `copyright` in table cells. The resulting XHTML document is shown in the screen capture at the end of Fig. 11.9. For more details on XSL, refer to Appendix D.
<tr>
    <!-- create table cells for ISBN in row 2 -->
    <td class = "bold">ISBN #:</td>
    <td><xsl:value-of select = "isbn" /></td>
</tr>
<tr>
    <!-- create table cells for edition in row 3 -->
    <td class = "bold">Edition:</td>
    <td><xsl:value-of select = "editionNumber" /></td>
</tr>
<tr>
    <!-- create table cells for copyright in row 4 -->
    <td class = "bold">Copyright:</td>
    <td><xsl:value-of select = "copyright" /></td>
</tr>
<tr>
    <!-- create Add to Cart button in row 5 -->
    <td>
        <form method = "post" action = "addToCart">
            <input type = "submit" value = "Add to Cart" />
        </form>
    </td>
</tr>
<tr>
    <!-- create View Cart button in row 5 -->
    <td>
        <form method = "get" action = "viewCart.jsp">
            <input type = "submit" value = "View Cart" />
        </form>
    </td>
</tr>
</table>

Fig. 11.9  XSL style sheet (books.xsl) that transforms a book's XML representation into an XHTML document (part 2 of 3).
11.6 Adding an Item to the Shopping Cart

When the user clicks the Add to Cart button in the XHTML document produced in the last section, the AddToCartServlet (aliased as addToCart) updates the shopping cart. If the cart does not exist, the servlet creates a shopping cart (a HashMap in this example). Items in the shopping cart are represented with CartItemBean objects. An instance of this JavaBean maintains a BookBean and the current quantity for that book in the shopping cart. When the user adds an item to the cart, if that item already is represented in the cart with a CartItemBean, the quantity of that item is updated in the bean. Otherwise, the servlet creates a new CartItemBean with a quantity of 1. After updating the cart, the user is forwarded to viewCart.jsp to view the current cart contents.

Class CartItemBean (Fig. 11.10) stores a BookBean and a quantity for that book. It maintains the BookBean as a read-only property of the bean and the quantity as a read–write property of the bean.

```java
1 // CartItemBean.java
2 // Class that maintains a book and its quantity.
3 package com.deitel.advjhtp1.store;
```

Fig. 11.9 XSL style sheet (books.xsl) that transforms a book’s XML representation into an XHTML document (part 3 of 3).
import java.io.*;

public class CartItemBean implements Serializable {
    private BookBean book;
    private int quantity;

    // initialize a CartItemBean
    public CartItemBean( BookBean bookToAdd, int number )
    {
        book = bookToAdd;
        quantity = number;
    }

    // get the book (this is a read-only property)
    public BookBean getBook()
    {
        return book;
    }

    // set the quantity
    public void setQuantity( int number )
    {
        quantity = number;
    }

    // get the quantity
    public int getQuantity()
    {
        return quantity;
    }
}

Fig. 11.10 CartItemBeans contain a BookBean and the quantity of a book in the shopping cart (part 2 of 2).

Class AddToCartServlet is shown in Fig. 11.11. The AddToCartServlet’s doGet method obtains the HttpSession object for the current client (line 18). If a session does not exist for this client, a RequestDispatcher forwards the request to the bookstore home page index.html (lines 22–26). Otherwise, line 29 obtains the value of session attribute cart—the Map that represents the shopping cart. Lines 30–31 obtain the value of session attribute bookToAdd—the BookBean representing the book to add to the shopping cart. If the shopping cart does not exist, lines 34–39 create a new HashMap to store the cart contents, then place the HashMap in the "cart" attribute of the session object. Lines 42–43 attempt to locate the CartItemBean for the book being added to the cart. If one exists, line 48 increments the quantity for that CartItemBean. Otherwise, line 50 creates a new CartItemBean with a quantity of 1 and puts it into the shopping cart (Map cart). Then lines 53–55 create a RequestDispatcher for JSP viewCart.jsp and forward the processing of the request to that JSP, so it can display the cart contents.
// AddToCartServlet.java
// Servlet to add a book to the shopping cart.
package com.deitel.advjhtp1.store;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;

public class AddToCartServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        HttpSession session = request.getSession( false );
        RequestDispatcher dispatcher;

        // if session does not exist, forward to index.html
        if ( session == null ) {
            dispatcher = request.getRequestDispatcher( "/index.html" );
            dispatcher.forward( request, response );
        }

        // session exists, get cart Map and book to add
        Map cart = ( Map ) session.getAttribute( "cart" );
        BookBean book = ( BookBean ) session.getAttribute( "bookToAdd" );

        // if cart does not exist, create it
        if ( cart == null ) {
            cart = new HashMap();

            // update the cart attribute
            session.setAttribute( "cart", cart );
        }

        // determine if book is in cart
        CartItemBean cartItem = ( CartItemBean ) cart.get( book.getISBN() );

        // If book is already in cart, update its quantity.
        // Otherwise, create an entry in the cart.
        if ( cartItem != null )
            cartItem.setQuantity( cartItem.getQuantity() + 1 );
        else
            cart.put( book.getISBN(), new CartItemBean( book, 1 ) );
    }
}

Fig. 11.11 AddToCartServlet places an item in the shopping cart and invokes viewCart.jsp to display the cart contents (part 1 of 2).
11.7 Viewing the Shopping Cart

JSP viewCart.jsp (Fi. 11.12) extracts the CartItemBeans from the shopping cart, subtotals each item in the cart, totals all the items in the cart and creates an XHTML document that allows the client to view the cart in tabular format. This JSP uses classes from our bookstore package (com.deitel.advjhtp1.store) and from packages java.util and java.text.

The scriptlet at lines 25–43 begins by retrieving the session attribute for the shopping cart Map (line 26). If there is no shopping cart, the JSP simply outputs a message indicating that the cart is empty. Otherwise, lines 34–41 create the variables used to obtain the information that is displayed in the resulting XHTML document. In particular, line 34 obtains the Set of keys in Map cart. These keys are used to retrieve the CartItemBean’s that represent each book in the cart.

Lines 45–51 output the literal XHTML markup that begins the table that appears in the document. Lines 55–63 continue the scriptlet with a loop that uses each key in the Map to obtain the corresponding CartItemBean, extracts the data from that bean, calculates the dollar subtotal for that product and calculates the dollar total of all products so far. The last part of the loop body appears outside the scriptlet at lines 70–86, in which the preceding data is formatted into a row in the XHTML table. JSP expressions are used to place each data value into the appropriate table cell. After the loop completes (line 90), lines 95–100 output the dollar total of all items in the cart and line 105 sets a session attribute containing the total. This value is used by process.jsp (Fig. 11.14) to display the dollar total as part of the order-processing confirmation. Line 101 outputs the dollar total of all items in the cart as the last row in the XHTML table.
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Shopping Cart</title>
<link rel="stylesheet" href="styles.css" type="text/css" />
</head>
<body>
<p class="bigFont">Shopping Cart</p>
<%-- start scriptlet to display shopping cart contents --%>

Map cart = (Map) session.getAttribute("cart");
double total = 0;
if ( cart == null || cart.size() == 0 )
  out.println("<p>Shopping cart is currently empty.</p>" );
else {
  // create variables used in display of cart
  Set cartItems = cart.keySet();
  Iterator iterator = cartItems.iterator();
  BookBean book;
  CartItemBean cartItem;
  int quantity;
  double price, subtotal;

  %> <%-- end scriptlet for literal XHTML output --%>

  <table>
    <thead><tr>
      <th>Product</th>
      <th>Quantity</th>
      <th>Price</th>
      <th>Total</th>
    </tr></thead>
    <tbody>
      while ( iterator.hasNext() ) {
        // get book data; calculate subtotal and total
        cartItem = (CartItemBean) cart.get( iterator.next() );
        book = cartItem.getBook();
        quantity = cartItem.getQuantity();
        price = book.getPrice();
        subtotal = quantity * price;
        total += subtotal;
      }
    </tbody>
  </table>
%
</body>
</html>

Fig. 11.12  JSP viewCart.jsp obtains the shopping cart and outputs an XHTML
document with the cart contents in tabular format (part 2 of 4).
<%-- end scriptlet for literal XHTML and --%>
<%-- JSP expressions output from this loop --%>
<%-- display table row of book title, quantity, --%>
<%-- price and subtotal --%>
<tr>
    <td><%= book.getTitle() %></td>
    <td><%= quantity %></td>
    <td class = "right">
        <%= new DecimalFormat( "0.00" ).format( price ) %>
    </td>
    <td class = "bold right">
        <%= new DecimalFormat( "0.00" ).format( subtotal ) %>
    </td>
</tr>

<% // continue scriptlet
} // end of while loop
<%-- end scriptlet for literal XHTML and --%>
<%-- display table row containing shopping cart total --%>
<tr>
    <td colspan = "4" class = "bold right">Total:
        <%= new DecimalFormat( "0.00" ).format( total ) %>
    </td>
</tr>
</table>

<% // continue scriptlet

// make current total a session attribute
session.setAttribute( "total", new Double( total ) );
} // end of else
<%-- end scriptlet --%>

<!-- link back to books.jsp to continue shopping -->
<p class = "bold green">
    <a href = "books.jsp">Continue Shopping</a>
</p>

Fig. 11.12 JSP viewCart.jsp obtains the shopping cart and outputs an XHTML document with the cart contents in tabular format (part 3 of 4).
From the XHTML document produced in this JSP, the user can either continue shopping or click the **Check Out** button to proceed to the *order.html* ordering page (Fig. 11.13).

### 11.8 Checking Out

When viewing the cart, the user can click a **Check Out** button to view *order.html* (Fig. 11.13). In this example, the form has no functionality. However, it is provided to help complete the application. Normally, there would be some client-side validation of the form elements, some server-side validation of form elements or a combination of both. When the user clicks the **Submit** button, the browser requests *process.jsp* to finalize the order.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- order.html -->
```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Order</title>
<link rel="stylesheet" href="styles.css" type="text/css" />
</head>
<body>
<p class="bigFont">Shopping Cart Check Out</p>
<p style="font-weight: bold">Please input the following information.</p>
<!-- table of form elements -->
<table>
<tr>
<td class="right bold">First name:</td>
<td>
<input type="text" name="firstname" size="25" />
</td>
</tr>
<tr>
<td class="right bold">Last name:</td>
<td>
<input type="text" name="lastname" size="25" />
</td>
</tr>
<tr>
<td class="right bold">Street:</td>
<td>
<input type="text" name="street" size="25" />
</td>
</tr>
<tr>
<td class="right bold">City:</td>
<td>
</td>
</tr>
</table>
</body>
</html>

Fig. 11.13 Order form (order.html) in which the user inputs name, address and credit-card information to complete an order (part 2 of 4).
<td>
  <input type = "text" name = "city" size = "25" />
</td>
</tr>
<tr>
  <td class = "right bold">State:</td>
  <td>
    <input type = "text" name = "state" size = "2" />
  </td>
</tr>
<tr>
  <td class = "right bold">Zip code:</td>
  <td>
    <input type = "text" name = "zipcode" size = "10" />
  </td>
</tr>
<tr>
  <td class = "right bold">Phone #:</td>
  <td>
    ( 
      <input type = "text" name = "phone" size = "3" />
    )
    <input type = "text" name = "phone2" size = "3" /> -
    <input type = "text" name = "phone3" size = "4" />
  </td>
</tr>
<tr>
  <td class = "right bold">Credit Card #:</td>
  <td>
    <input type = "text" name = "creditcard" size = "25" />
  </td>
</tr>
<tr>
  <td class = "right bold">Expiration (mm/yy):</td>
  <td>
    <input type = "text" name = "expires" size = "2" />
  </td>
</tr>

Fig. 11.13 Order form (order.html) in which the user inputs name, address and credit-card information to complete an order (part 3 of 4).
<input type="text" name="expires2" size="2" />
</td>
</tr>
</table>

<!-- enable user to submit the form -->
<p><input type="submit" value="Submit" /></p>
</form>
</body>
</html>

Fig. 11.13  Order form (order.html) in which the user inputs name, address and credit-card information to complete an order (part 4 of 4).
11.9 Processing the Order

JSP process.jsp (Fig. 11.14) pretends to process the user’s credit-card information and creates an XHTML document containing a message that the order was processed and the final order dollar total. The scriptlet at lines 19–28 obtains the session attribute total. The Double object returned is converted to a double and stored in Java variable total. Our simulation of a bookstore does not perform real credit-card processing, so the transaction is now complete. Therefore, line 26 invokes HttpSession method invalidate to discard the session object for the current client. In a real store, the session would not be invalidated until the purchase is confirmed by the credit-card company. Lines 30–40 define the body of the XHTML document sent to the client. Line 37 uses a JSP expression to insert the dollar total of all items purchased.

```xml
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- process.jsp -->

<%@ page language = "java" session = "true" %>
<%@ page import = "java.text.*" %>

<html xmlns = "http://www.w3.org/1999/xhtml">
<head>
<title>Thank You!</title>
<link rel = "stylesheet" href = "styles.css"
type = "text/css" />
</head>

<!-- start scriptlet

// get total order amount
Double d = ( Double ) session.getAttribute( "total" );
double total = d.doubleValue();
// invalidate session because processing is complete
session.invalidate();

--> <!-- end scriptlet -->

<body>
<p class = "bigFont">Thank You</p>
<p>Your order has been processed.</p>
<p>Your credit card has been billed:
   $<%= new DecimalFormat( "0.00" ).format( total ) %>
</p>
</body>
</html>
```

Fig. 11.14  JSP process.jsp performs the final order processing (part 1 of 2).
11.10 Deploying the Bookstore Application in J2EE 1.2.1

Next, we deploy the bookstore application in the Java 2 Enterprise Edition 1.2.1 reference implementation. This section assumes that you have downloaded and installed J2EE 1.2.1. If not, please refer to Appendix E for installation and configuration instructions. Note that the files for this entire bookstore application can be found on the CD that accompanies this book and at www.deitel.com.

In Section 11.10.1 through Section 11.10.8, we demonstrate the steps needed to deploy this application:

1. Configure the books data source for use with the J2EE 1.2.1 reference implementation server.
2. Launch the Cloudscape database server and the J2EE 1.2.1 reference implementation server for deployment and execution of the application.
3. Launch the Application Deployment Tool. This tool provides a graphical user interface for deploying applications on the J2EE 1.2.1 server.
4. Create a new application in the Application Deployment Tool.
5. Add library JAR files to the application. These files are available to all application components.
6. Create a new Web component in the application for the BookServlet.
7. Create a new Web component in the application for the AddToCartServlet.
8. Add the nonservlet components to the application. These include XHTML documents, JSPs, images, CSS files, XSL files and JavaBeans used in the application.
9. Specify the Web context that causes this J2EE application to execute. This determines the URL that will be used to invoke the application.
10. Specify the database resource (i.e., **books**) used by our application.

11. Set up the JNDI name for the database in the application. This is used to register the name with the Java Naming and Directory Service so the database can be located at execution time.

12. Set up the **welcome file** for the application. This is the initial file that is returned when the user invokes the bookstore application.

13. Deploy the application.

14. Run the application.

At the end of Section 11.10.8, you will be able to deploy and test the bookstore application.

### 11.10.1 Configuring the **books** Data Source

Before deploying the bookstore application, you must configure the **books** data source so the J2EE server registers the data source with the naming server. This enables the application to use JNDI to locate the data source at execution time. J2EE comes with Cloudscape—a pure-Java database application from Informix Software. We use Cloudscape to perform our database manipulations in this case study.

To configure the Cloudscape data source, you must modify the J2EE default configuration file `default.properties` in the J2EE installation’s `config` directory. Below the comment `JDBC URL Examples` is a line that begins with `jdbc.datasources`. Append the following text to this line

```
jdbc/books|jdbc:cloudscape:rmi:books;create=true
```

The vertical bar, |, at the beginning of the text separates the new data source we are registering from a data source that is registered by default when you install J2EE. The text `jdbc/books` is the JNDI name for the database. After the second | character in the preceding text is the JDBC URL `jdbc:cloudscape:rmi:books`. The URL indicates that the J2EE will use the JDBC protocol to interact with the Cloudscape subprotocol, which, in turn, uses RMI to communicate with the database (**books** in this case). Finally, `create=true` specifies that J2EE should create a database if the database does not already exist. [Remember, that the **books** database was created in Chapter 8.] After configuring the database, save the `default.properties` file. This completes Step 1 of Section 11.10.

### Portability Tip 11.2

Each database driver typically has its own URL format that enables an application to interact with databases hosted on that database server. See your database server’s documentation for more information.

### 11.10.2 Launching the Cloudscape Database and J2EE Servers

**Step 2** of Section 11.10 specifies that you must launch the Cloudscape database server and the J2EE server, so you can deploy and execute the application. First, open a command prompt (or shell) and launch the Cloudscape server as discussed in Section 8.5. Next, open
a command prompt (or shell) and change directories to the `bin` subdirectory of your J2EE
installation. Then, issue the following command:

```
j2ee -verbose
```

to start the J2EE server. Note that the J2EE server includes the Tomcat JSP and servlet con-
tainer discussed in Chapter 9.

### Portability Tip 11.3

On some UNIX/Linux systems, you may need to precede the commands that launch the
Cloudscape server and the J2EE server with `./`, to indicate that the command is located in
the current directory.

### Testing and Debugging Tip 11.1

Use separate command prompts (or shells) to execute the commands that launch the Clouds-
cape database server and the J2EE 1.2.1 server, so you can see any error messages gener-
at by these programs.

### Testing and Debugging Tip 11.2

To ensure that the J2EE server communicates properly with the Cloudscape server (or any
other database server), always launch the database server before the J2EE server. Other-
wise, exceptions will occur when the J2EE server attempts to configure its data sources.

To shut down the J2EE server, use a command prompt (or shell) to execute the fol-
lowing command from the `bin` subdirectory of your J2EE installation:

```
j2ee -stop
```

### Testing and Debugging Tip 11.3

Always shut down the J2EE server before the Cloudscape database server to ensure that the
J2EE server does not attempt to communicate with the Cloudscape database server after the
database server has been shut down. If Cloudscape is terminated first, it is possible that the
J2EE server will receive another request and attempt to access the database again. This will
result in exceptions.

#### 11.10.3 Launching the J2EE Application Deployment Tool

Step 3 of Section 11.10 begins the process of deploying our bookstore application. The
J2EE reference implementation comes with a graphical application, called the Application
Deployment Tool, that helps you deploy Enterprise Java applications. In Chapter 9,
we created an XML deployment descriptor by hand to deploy our servlets. The Application
Deployment Tool is nice in that it writes the deployment descriptor files for you and
automatically archives the Web application’s components. The tool places all Web appli-
cation components and auxiliary files for a particular application into a single Enterprise
Application Archive (EAR) file. This file contains deployment descriptor information,
WAR files with Web application components and additional information that is discussed
later in the book.

Execute the deployment tool by opening a command prompt (or shell) and changing
directories to the `bin` subdirectory of your J2EE installation. Then, type the following
command:
deploytool

The Application Deployment Tool window (Fig. 11.15) appears. [Note: In our deployment discussion, we cover only those aspects of the deployment tool required to deploy this bookstore application. Later in the book, we discuss other aspects of this tool in detail.]

11.10.4 Creating the Bookstore Application

The Application Deployment Tool simplifies the task of deploying Enterprise applications. Next (Step 4 of Section 11.10), we create the new application. Click the New Application button to display the New Application window (Fig. 11.16).

In the Application File Name field, you can type the name of the EAR file in which the Application Deployment Tool stores the application components, or you can click Browse to specify both the name and location of the file. In the Application Display Name field, you can specify the name for your application. This name will appear in the Local Applications area of the deployment tool’s main window (Fig. 11.15). Click OK to create the application. The main Application Deployment Tool window now appears as shown in Fig. 11.17.

---

![Fig. 11.15 Application Deployment Tool main window.](image)

![Fig. 11.16 New Application window.](image)
11.10.5 Creating BookServlet and AddToCartServlet Web Components

Step 6 of Section 11.10 is to create Web components for the BookServlet and the AddToCartServlet. This will enable us to specify the alias that is used to invoke each servlet. We will show the details of creating the BookServlet Web component. Then, you can repeat the steps to create the AddToCartServlet Web component.

To begin, click the New Web Component button (see Fig. 11.15) to display the Introduction window of the New Web Component Wizard (Fig. 11.18).

Click the Next > button to display the WAR File General Properties (Fig. 11.19) window of the New Web Component Wizard.

Ensure that JSP and Servlet Bookstore is selected in the Web Component will Go In: drop-down list. In the WAR Display Name field, type a name (Store Components) for the WAR that will appear in the Local Applications area of the deployment tool’s main window (see Fig. 11.15). Then, click the Add... button to display the Add Files to .WAR - Add Content Files window (Fig. 11.20). Content files are nonservlet files such as images, XHTML documents, style sheets and JSPs. We will be adding these in another step later, so click Next > to proceed to the Add Files to .WAR - Add Class Files window (Fig. 11.21)
Fig. 11.18 **New Web Component Wizard - Introduction** window.

Fig. 11.19 **New Web Component Wizard - WAR File General Properties** window.
To add the `BookServlet.class` file, click the **Browse...** button to display the **Choose Root Directory** window (Fig. 11.22).
When adding a class file for a class in a package (as all classes are in this example), it is imperative that the files be added and maintained in their full package directory structure or as part of a JAR file that contains the full package directory structure. In this example, we did not create a JAR file containing the entire `com.deitel.advjhtp1.store` package. Therefore, we need to locate the directory in which the first package directory name is found.

On our system, the `com` directory that starts the package name is located in a directory called `Development`. When you click the *Choose Root Directory* button, you are returned to the *Add Files to .WAR - Add Class Files* window. In that window, you should be able to locate the `com` directory (Fig. 11.23).

---

**Fig. 11.22** Choose Root Directory window.

**Fig. 11.23** Add Files to .WAR - Add Class Files window after selecting the root directory in which the files are located.
Double click the `com` directory name to expand its contents in the window. Do the same for the subdirectory `deitel`, then the subdirectory `advjhtp1` and finally, for the directory `store`. In the `store` directory, select the `.class` file for `BookServlet`, then click the `Add` button. At the bottom of the `Add Files to .WAR - Add Class Files` window, the `.class` file should be displayed with its full package directory structure. After doing this, click the `Finish` button to return to the `New Web Component Wizard - WAR File General Properties` window. Note that the files selected with the `Add Files to .WAR` window now appear in the `Contents` text area of the window (Fig. 11.24).

**Common Programming Error 11.1**

Not including the full package directory structure for a class in a package will prevent the application from loading the class and from executing properly.

Click `Next >` to proceed to the `New Web Component Wizard - Choose Component Type` window and select `Servlet` (Fig. 11.25).

Click `Next >` to proceed to the `New Web Component Wizard - Component General Properties` window (Fig. 11.26). Select the `BookServlet` class in the `Servlet Class` drop-down list and type `BookServlet` in the `Web Component Display Name` field.
Click **Next >** twice to display the **New Web Component Wizard - Component Aliases** window (Fig. 11.27). Click **Add** to specify an alias for the **BookServlet**. Click the white box that appears in the window, type **displayBook** as the alias for the servlet and press **Enter**. Next, click **Finish** to complete the setup of the **BookServlet**.
Now, create a Web component for `AddToCartServlet` (Step 7 of Section 11.10), by repeating the steps shown in this section. For this Web component, specify `Add to Cart Servlet` as the Web Component Display Name and `addToCart` as the alias for the servlet. After adding the two servlet Web components, the Application Deployment Tool window should appear as shown in Fig. 11.28.

### 11.10.6 Adding Non-Servlet Components to the Application

Next, we will add our non-servlet components to the application (Step 8 of Section 11.10). These files include JSPs, XHTML documents, style sheets, images and JavaBeans used in the application.

Begin by expanding the application component tree and clicking Store Components in the Local Applications area of the Application Deployment Tool window (see Fig. 11.28). In the contents area of the Application Deployment Tool window, click the Add... button to display the Add Files to .WAR - Add Content Files window (Fig. 11.29).
Navigate to the directory on your system that contains the files for the bookstore application. In the list box that appears in the window, locate each of the following files and directories: `book.xsl`, `books.jsp`, `images`, `index.html`, `order.html`, `process.jsp`, `styles.css` and `viewCart.jsp`. For each file or directory, click the **Add** button. You can select multiple items at one time by holding down the **<Ctrl>** key and clicking each item. All the items you add should appear in the text area at the bottom of the window. When you are done, click **Next >** to display the **Add Files to .WAR - Add Class Files** window (Fig. 11.30).

We will use this window to add the **.class** files for the non-servlet classes (i.e., our JavaBeans) to our application. Remember that the JavaBeans used in the bookstore are in a package, so their **.class** files must be added and maintained in their full package directory structure. Once again, click the **Browse...** button to display the **Choose Root Directory** window and locate the directory in which the first package directory name is found. Select that directory as the root directory. Double click the **com** directory name to expand its contents in the window. Do the same for the subdirectory **deitel**, then the subdirectory **advjhtp1** and finally, for the directory **store**. In the **store** directory, select the **.class** files for each of the JavaBeans in this bookstore example (**BookBean.class**, **CartItemBean.class** and **TitlesBean.class**), then click the **Add** button. At the bottom of the **Add Files to .WAR - Add Class Files** window, each **.class** file should be displayed with its full package directory structure. After doing this, click the **Finish** button to return to the **Application Deployment Tool** window. Note that the files selected with the **Add Files to .WAR** windows now appear in the **Contents** text area of the window. Click the **Save** button to save your work.
11.10.7 Specifying the Web Context, Resource References, JNDI Names and Welcome Files

Steps 9 through 13 of Section 11.10 perform the final configuration and deployment of the bookstore application. After performing the steps in this section, you will be able to execute the bookstore application.

We begin by specifying the Web context for our application (Step 9 of Section 11.10). At the beginning of this chapter, we indicated that the user would enter the URL

http://localhost:8000/advjhtp1/store

in a browser to access the bookstore application. The Web context is the part of the preceding URL that enables the server to determine which application to execute when the server receives a request from a client. In this case, the Web context is advjhtp1/store. Once again, note that the J2EE server uses port 8000, rather than port 8080, used by Tomcat.

**Common Programming Error 11.2**

Specifying the wrong port number in a URL that is supposed to access the J2EE server causes your Web browser to indicate that the server was not found.

**Testing and Debugging Tip 11.4**

When deploying an Enterprise Java application on a production application server (the J2EE server is for testing only), it is typically not necessary to specify a port number in the URL when accessing the application. See your application server’s documentation for further details.

To specify the Web context, click the JSP and Servlet Bookstore node in the Local Applications area of the Application Deployment Tool window. Then, click
the **Web Context** tab (Fig. 11.31). Click the white box in the **Context Root** column and type `advjhttp1/store`; then click Enter.

Next, we must specify the database resource referenced by the bookstore application (Step 10 of Section 11.10). Click the **Store Components** node in the **Local Applications** area of the **Application Deployment Tool** window. Then, click the **Resource Ref's** tab (Fig. 11.32). Click the **Add** button. Under the **Coded Name** column click the white box and type `jdbc/books` (the JNDI name of our data source). Figure 11.32 shows the **Application Deployment Tool** window after creating the resource reference.

Next, we specify the JNDI name for the database in the application (Step 11 of Section 11.10). This is used to register the name with the Java Naming and Directory Service, so the database can be located by the application at execution time.

---

**Fig. 11.31** Specifying the **Web Context** in the **Application Deployment Tool**.

**Fig. 11.32** Specifying the **Resource Ref's** in the **Application Deployment Tool**.
To specify the JNDI name for the database, click the JSP and Servlet Bookstore node in the Local Applications area of the Application Deployment Tool window. Then, click the JNDI names tab (Fig. 11.33). In the JNDI Name column, click the white box and type jdbc/books; then click Enter.

The last task to perform before deploying the application is specifying the welcome file that is displayed when the user first visits the bookstore. Click the Store Components node in the Local Applications area of the Application Deployment Tool window. Then, click the File Ref’s tab (Fig. 11.34). Click the Add button. In the Welcome Files area click the white box and type index.html. Figure 11.34 shows the Application Deployment Tool window after specifying the welcome file. Click the Save button to save the application settings.

---

**Fig. 11.33** Specifying the Resource Ref’s in the Application Deployment Tool.

**Fig. 11.34** Specifying the welcome file in the File Ref’s tab of the Application Deployment Tool.
11.10.8 Deploying and Executing the Application

Now, you are ready to deploy the bookstore application so you can test it. Figure 11.35 shows Application Deployment Tool toolbar buttons for updating application files and deploying applications. The Update Application Files button updates the application’s EAR file after changes are made to any of the files, such as recompiling classes or modifying files. The Deploy Application button causes the Application Deployment Tool to communicate with the J2EE server and deploy the application. The functionality of both these buttons is combined in the Update and Redeploy Application button.

Click the Deploy Application button to display the Deploy JSP and Servlet Bookstore - Introduction window (Fig. 11.36). Click the Next > button three times, then click the Finish button. A Deployment Progress window appears. This window will indicate when the deployment is complete. When that occurs, open a Web browser and enter the following URL to test the bookstore application:

```
http://localhost:8000/advjhttp1/store
```

Fig. 11.35 Application Deployment Tool toolbar buttons for updating application files and deploying applications.

Fig. 11.36 Deploy JSP and Servlet Bookstore - Introduction window.
In this chapter, we have presented our first substantial Enterprise Java application. The steps presented in this section for deploying the bookstore application are just some of the steps required in a typical Enterprise application. For example, there were no security requirements in the bookstore application. In real Enterprise Java applications, some or all of the application components have security restrictions, such as “the user must enter a valid username and password before access is granted to a component.” Such restrictions are specified at deployment time with the **Application Deployment Tool** or some similar tool in an Enterprise Java development environment. These security restrictions are enforced by the application server. In our bookstore example, if the JSPs had security restrictions, it would be necessary to deploy each one individually, as we did with the **BookServlet** and the **AddToCartServlet**. In later chapters, we discuss more of the deployment options for application components. The *Java 2 Enterprise Edition Specification* (available at [java.sun.com/j2ee/download.html](http://java.sun.com/j2ee/download.html)) discusses the complete set of deployment options that are required in J2EE-compliant application servers.

The next chapter continues our client/server discussions. In that chapter, we use servlets and XML to create content for wireless devices, such as pagers, cell phones and personal digital assistants.

**SUMMARY**

- The Java 2 Enterprise Edition 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container.
- A three-tier, distributed Web application consists of client, server and database tiers.
- The client tier in a Web application often is represented by the user’s Web browser.
- The server tier in a Web application often consists of JSPs and servlets that act on behalf of the client to perform tasks.
- The database tier maintains the database accessed from the server tier.
- The Java 2 Enterprise Edition 1.2.1 reference implementation comes with an **Application Deployment Tool**. Among its many features, this tool enables us to specify the alias used to invoke a servlet.
- The **Application Deployment Tool** creates the deployment descriptor for a servlet as part of deploying an application.
- The welcome file is the default document sent as the response to a client when the client initially interacts with a J2EE application.
- Different browsers have different levels of support for Cascading Style Sheets.
- JavaServer Pages often generate XHTML that is sent to the client for rendering.
- The Java Naming and Directory Interface (JNDI) enables Enterprise Java application components to access information and resources (such as databases) that are external to an application. In some cases, those resources are distributed across a network.
- An Enterprise Java application container must provide a naming service that implements JNDI and enables the components executing in that container to perform name lookups to locate resources. The J2EE 1.2.1 reference implementation server includes such a naming service.
- When you deploy an Enterprise Java application, you specify the resources used by the application (such as databases) and the JNDI names for those resources.
- Using an **InitialContext**, an Enterprise application component can look up a resource. The **InitialContext** provides access to the application’s naming environment.
• **InitialContext** method `lookup` locates a resource with a JNDI name. Method `lookup` returns an `Object` object and throws a `NamingException` if it cannot resolve the name it receives as an argument.

• A **DataSource** is used to connect to a database.

• The `org.w3c.dom` package’s **Document** and **Element** interfaces are used to create an XML document tree.

  • **Document** method `createElement` creates an element for an XML document.
  
  • **Document** method `createTextNode` specifies the text for an **Element**.
  
  • **Element** method `appendChild` appends a node to an **Element** as a child of that **Element**.
  
• An XML document can be transformed into an XHTML document using an XSL style sheet.

• **ServletRequest** method `getRequestDispatcher` returns a **RequestDispatcher** object that can `forward` requests to other resources or `include` other resources as part of the current servlet’s response.

• When **RequestDispatcher** method `forward` is called, processing of the request by the current servlet terminates.

• **RequestDispatcher** objects can be obtained with method `getRequestDispatcher` from an object that implements interface **ServletRequest**, or from the **ServletContext** with methods `getRequestDispatcher` or `getNamedDispatcher`.

• **ServletContext** method `getNamedDispatcher` receives the name of a servlet as an argument, then searches the **ServletContext** for a servlet by that name. If no such servlet is found, the method returns `null`.

• Both the **ServletRequest** and the **ServletContext** `getRequestDispatcher` methods simply return the content of the specified path if the path does not represent a servlet.

• Before the XML and XSL capabilities can be used, you must download and install Sun’s Java API for XML Parsing (JAXP) version 1.1 from [java.sun.com/xml/download.htm](http://java.sun.com/xml/download.htm).

• The root directory of JAXP (normally called `jaxp-1.1`) contains three JAR files that are required for compiling and running programs that use JAXP—`crimson.jar`, `jaxp.jar` and `xalan.jar`. These files must be added to the Java extension mechanism for your Java 2 Standard Edition installation.

• JAXP 1.1 is part of the forthcoming J2EE 1.3 reference implementation.

• Creating a Document Object Model (DOM) tree from an XML document requires a **DocumentBuilder** parser object. **DocumentBuilder** objects are obtained from a **DocumentBuilderFactory**.

• Classes **Document** and **Element** are located in package `org.w3c.dom`.

• A **DOMSource** represents an XML document in an XSL transformation. A **StreamSource** can be used to read a stream of bytes that represent an XSL file.

• A **StreamResult** specifies the **PrintWriter** to which the results of the XSL transformation are written.

• **TransformerFactory** static method `newInstance` creates a **TransformerFactory** object. This object enables the program to obtain a **Transformer** object that applies the XSL transformation.

• **TransformerFactory** method `newTransformer` receives a **StreamSource** argument representing the XSL that will be applied to an XML document.

• **Transformer** method `transform` performs an XSL transformation on the given **DOMSource** object and writes the result to the given **StreamResult** object.
A TransformerException is thrown if a problem occurs when creating the TransformerFactory, creating the Transformer or performing the transformation.

 HttpSession method invalidate discards the session object for the current client.

 Before deploying an Enterprise Java application, you must configure your data sources and other resources, so the J2EE server can register those resources with the naming server. This enables the application to use JNDI to locate the resources at execution time.

 J2EE comes with Cloudscape—a pure-Java database application from Informix Software.

 To configure a Cloudscape data source, you must modify the J2EE default configuration file default.properties in the J2EE installation’s config directory. Below the comment JDBC URL Examples is a line that begins with jdbc.datasources. Append the following text (in which dataSource represents your data source name) to this line:

 |jdbc/dataSource|jdbc:cloudscape:rmi: dataSource;create=true

 Each database server typically has its own URL format that enables an application to interact with databases hosted on that database server.

 You must launch the Cloudscape database server and the J2EE server before you can deploy and execute an application.

 To ensure that the J2EE server communicates properly with the Cloudscape server (or any other database server), always launch the database server before the J2EE server.

 Always shut down the J2EE server before the Cloudscape database server, to ensure that the J2EE server does not attempt to communicate with the Cloudscape database server after the database server has been shut down.

 The J2EE reference implementation comes with a graphical application, called the Application Deployment Tool, that helps you deploy Enterprise Java applications.

 The Application Deployment Tool is nice in that it writes the deployment descriptor files for you and automatically archives the Web application’s components. The tool places all Web application components and auxiliary files for a particular application into a single Enterprise Application Archive (EAR) file. This file contains deployment descriptor information, WAR files with Web application components and additional information that is discussed later in the book.

 When adding a class file for a class in a package to an application with the Application Deployment Tool, it is imperative that the files be added and maintained in their full package directory structure or as part of a JAR file that contains the full package directory structure.

 Not including the full package directory structure for a class in a package will prevent the application from loading the class and from executing properly.

 The Web context for an application is the part of the URL that enables the server to determine which application to execute when the server receives a request from a client.

 The J2EE server uses port 8000, rather than port 8080, used by Tomcat.

 When deploying an Enterprise Java application on a production application server, it is typically not necessary to specify a port number in the URL when accessing the application.

 Part of deploying an application is to specify the resource references for the components in the application.

 Each resource reference has a corresponding JNDI name that is used by the deployment tool to register the resource with the Java Naming and Directory Service. This enables the resource to be located by the application at execution time.
**TERMINOLOGY**

**Application Deployment Tool**
Cascading Style Sheets (CSS)
component environment entries
configure a data source
create a Web component
database resource
**DataSource** interface
describe properties
deploy an application
dynamic XHTML document
Enterprise Application Archive (EAR) file
external resource
**forward** method of **RequestDispatcher**
**getRequestDispatcher** method of **ServletRequest**
include content from a resource
**include** method of **RequestDispatcher**
**InitialContext** class
**invalidate** method of **HttpServletRequest**
J2EE **config** directory
Java 2 Enterprise Edition 1.2.1
reference implementation
Java API for XML Parsing (JAXP)
Java Naming and Directory Interface (JNDI)
**javax.naming** package
**jdbc.datasources** J2EE
configuration property
**jdbc:cloudscape:rmi:books** JDBC URL
JNDI name
locate a naming service
**lookup** method of **InitialContext**
named lookup
name resolution
naming service
register a data source with a naming server
**RequestDispatcher** interface
server tier
**ServletContext** interface
shopping cart
style sheet
Web component
Web context
welcome file
XML
XSL transformation

**SELF-REVIEW EXERCISES**

11.1 Fill in the blanks in each of the following statements:

- a) A three-tier, distributed Web application consists of ________, ________ and ________ tiers.
- b) The ________ is the default document sent as the response to a client when the client initially interacts with a J2EE application.
- c) The ________ enables Enterprise Java application components to access information and resources (such as databases) that are external to an application.
- d) An ________ object provides access to the application’s naming environment.
- e) A **RequestDispatcher** object can ________ requests to other resources or ________ other resources as part of the current servlet’s response.
- f) Sun’s ________ provides XML and XSL capabilities in a Java program.
- g) Method ________ of interface ________ discards the session object for the current client.
- h) The ________ for an application is the part of the URL that enables the server to determine which application to execute when the server receives a request from a client.
- i) An Enterprise Java application container must provide a ________ that implements JNDI and enables the components executing in that container to perform name lookups to locate resources.
- j) The J2EE reference implementation comes with a graphical application, called the ________, that helps you deploy Enterprise Java applications.

11.2 State whether each of the following is true or false. If false, explain why.

- a) The J2EE server uses port 8080 to await client requests.
b) When deploying applications with the J2EE server, you can launch the Cloudscape and J2EE servers in any order.

c) **InitialContext** method `lookup` locates a resource with a JNDI name.

d) Method `lookup` returns a **Connection** object representing the connection to the database.

e) The Java 2 Enterprise Edition 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container.

f) When `RequestDispatcher` method `forward` is called, processing of the request by the current servlet is temporarily suspended to wait for a response from the resource to which the request is forwarded.

```
g) Both the `ServletRequest` and the `ServletContext` `getRequestDispatcher` methods throw exceptions if the argument to `getRequestDispatcher` is not a servlet.
```

h) Each resource reference has a corresponding JNDI name that is used by the deployment tool to register the resource with the Java Naming and Directory Service.

i) If you do not configure your data sources and other resources before deploying an Enterprise Java application, the J2EE server will search the application to determine the resources used and register those resources with the naming server.

j) Not including the full package directory structure for a class in a package will prevent the application from loading the class and from executing properly.

**ANSWERS TO SELF-REVIEW EXERCISES**

11.1 a) client, server, database.  
b) welcome file.  
c) Java Naming and Directory Interface (JNDI).  
d) `InitialContext`.  
e) `forward`, `include`.  
f) Java API for XML Parsing (JAXP).  
g) `invalidate`, `HttpSession`.  
h) Web context.  
i) naming service.  
j) Application Deployment Tool.

11.2 a) False. Port 8080 is the default port for the Tomcat server. The J2EE server uses port 8000.

```
b) False. To ensure that the J2EE server communicates properly with the Cloudscape server (or any other database server), the database server must be launched before the J2EE server.
```

c) True.

```
d) False. Method `lookup` returns a `DataSource` object that can be used to obtain a `Connection`.
```

e) True.

```
f) False. When `RequestDispatcher` method `forward` is called, processing of the request by the current servlet terminates.
```

```
g) False. Both the `ServletRequest` and the `ServletContext` `getRequestDispatcher` methods simply return the contents of the specified path if the path does not represent a servlet.
```

```
h) True.
```

```
i) False. Before deploying an Enterprise Java application, you must configure your data sources and other resources, so the J2EE server can register those resources with the naming server. Otherwise, exceptions will occur when the application attempts to access the resources.
```

```
j) True.
```

**EXERCISES**

11.3 Modify the bookstore case study to enable the client to change the quantity of an item currently in the shopping cart. In `viewCart.jsp`, display the quantity in an `input` element of type
text in a form. Provide the user with a submit button with the value Update Cart that enables
the user to submit the form to a servlet that updates the quantity of the items in the cart. The servlet
should forward the request to viewCart.jsp, so the user can see the updated cart contents. Redeploy
the bookstore application, and test the update capability.

11.4 Enhance the bookstore case study’s TitlesBean to obtain author information from the
books database. Incorporate that author data into the BookBean class, and display the author informa-
tion as part of the Web page users see when they select a book and view that book’s information.

11.5 Add server-side form validation to the order form in the bookstore case study. Check that the
credit-card expiration date is after today’s date. Make all the fields in the form required fields. When
the user does not supply data for all fields, return an XHTML document containing the order form.
Any fields in which the user previously entered data should contain that data. For this exercise, re-
place the order.html document with a JSP that generates the form dynamically.

11.6 Create an order table and an orderItems table in the books database to store orders
placed by customers. The order table should store an orderID, an orderDate and the email
address of the customer who placed the order. [Note: You will need to modify the form in
Exercise 11.6 to include the customer’s e-mail address]. The orderItems table should store the
orderID, ISBN, price and quantity of each book in the order. Modify process.jsp so that
it stores the order information in the order and orderItems tables.

11.7 Create a JSP that enables client to view their order history. Integrate this JSP into the book-
store case study.

11.8 Create and deploy a single application that allows a user to test all the JSP examples in
Chapter 10. The application should have a welcome file that is an XHTML document containing links
to each of the examples in Chapter 10.

11.9 Create and deploy a single application that allows a user to test all the servlets in Chapter 9.
The application should have a welcome file that is an XHTML document containing links to each of
the examples in Chapter 9.
Objectives

- To construct a three-tier, client/server application.
- To use XML and XSLT to present content for several client types.
- To understand the Java 2 Micro Edition (J2ME) Platform.
- To understand the MIDlet lifecycle.
- To be able to use J2ME CLDC and MIDP.
- To understand how our case study incorporates J2ME technology.

One thing I know: the only ones among you who will be really happy are those who will have sought and found how to serve.
Albert Schweitzer

It was a miracle of rare device, …
Samuel Taylor Coleridge

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it.
Samuel Johnson

… all the light I can command must be concentrated on this particular web …
George Elliot

When you can do the common things of life in an uncommon way, you will command the attention of the world.
George Washington Carver
12.1 Introduction

In this chapter, present a case study of a multiple-choice test (*Tip Test*) for users to test their knowledge of Deitel programming tips. Each question consists of a tip image and a list of four possible answers. A client downloads the test from a server. The user then selects an answer and submits it to the server, which responds with content that describes if the answer is correct or incorrect. The client then can download another question and continue playing indefinitely.

The Tip-Test application is a three-tier architecture, as shown in Fig. 12.1. The information tier consists of a database that contains a table (defined in the SQL script `tips.sql`) with seven rows and five columns. Each row contains information about a Deitel programming tip—Good Programming Practice, Software-Engineering Observation, Performance Tip, Portability Tip, Look-And-Feel Observation, Testing and Debugging Tip and Common Programming Error. The first database column stores integers that represent unique identifiers for each tip. The second column stores the names of the tips. The third column stores the tips’ descriptions—i.e., definitions of each tip and explanations of why each tip is important. The fourth column stores the image names for each tip. The fifth column stores the tips’ abbreviated names—e.g., the abbreviation for Good Programming Practice is GPP. Figure 12.2 shows the contents of `tips.sql`.

The middle tier consists of two servlets—*WelcomeServlet* and *TipTestServlet*. *WelcomeServlet* delivers a “welcome screen” that introduces the game to the user. *WelcomeServlet* then redirects the client to *TipTestServlet*. Using the database, *TipTestServlet* randomly selects a tip image and four possible answers (in the form of abbreviated tip names) and marks up this information as an XML document. *TipTestServlet* then applies an XSL transformation to the XML document and sends the resulting content to the client.
Fig. 12.1 Three-tier architecture for Tip Test.

<table>
<thead>
<tr>
<th>tipID</th>
<th>tipName</th>
<th>tipDescription</th>
<th>tipImage</th>
<th>shortName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Good Programming Practice</td>
<td>Good Programming Practice call the student’s attention to techniques that...</td>
<td>goodProgramming</td>
<td>GPP</td>
</tr>
<tr>
<td>2</td>
<td>Common Programming Error</td>
<td>Students learning a language tend to make certain kinds of errors...</td>
<td>programmingError</td>
<td>CPE</td>
</tr>
<tr>
<td>3</td>
<td>Look-and-Feel Observation</td>
<td>We provide Look-And-Feel Observations to highlight graphical user...</td>
<td>lookAndFeel</td>
<td>LAF</td>
</tr>
<tr>
<td>4</td>
<td>Performance Tip</td>
<td>Performance Tips highlight opportunities for improving program performance.</td>
<td>perf</td>
<td>PERF</td>
</tr>
<tr>
<td>5</td>
<td>Portability Tip</td>
<td>Organizations that develop software must often produce versions customized...</td>
<td>portability</td>
<td>PORT</td>
</tr>
</tbody>
</table>

Fig. 12.2 Database contents of tips.sql (part 1 of 2).
The client tier consists of four client types—Internet Explorer, WAP (Wireless Application Protocol), i-mode and J2ME. Each client can render a different content type. **TipTestServlet** handles all game logic—i.e., selecting the Tip-Test question at random and determining if the user’s answer is correct—and sends the content to each client. Microsoft Internet Explorer receives XHTML (Extensible HyperText Markup Language) content. The Openwave UP simulator is the WAP client that receives WML (Wireless Markup Language) content. WAP is a protocol that enables wireless devices to transfer information over the Internet. WML marks up content rendered on the wireless device. The Pixo Internet Microbrowser is the i-mode client that receives cHTML content—i-mode provides a popular Japanese-based wireless Internet service, and cHTML (compact HTML) is an HTML subset for resource-limited devices. The Sun MIDP-device emulator acts as the J2ME client that receives content in plain-text format. J2ME™ (Java™ 2 Micro Edition) is Sun’s newest Java platform for developing applications for various consumer devices, such as set-top boxes, Web terminals, embedded systems, mobile phones and cell pagers. MIDP (Mobile Information Device Profile) is a set of APIs that allows developers to handle mobile-device-specific issues, such as creating user interfaces, storing information locally and networking. Devices that run applications for MIDP are called **MIDP** devices (e.g., cell phones or pagers). We discuss J2ME and MIDP in much greater detail in Section 12.4.

We present this case study by discussing how each servlet sends content to each client type, then how each client type renders that content. Section 12.2 and Section 12.3 discuss how **WelcomeServlet** and **TipTestServlet** handle client requests. Sections 27.3.1–27.3.4 discuss how **TipTestServlet** responds to each client request and how each client renders the servlet-generated content. Lastly, Section 12.4 discusses Java 2 Micro Edition. Our treatment of J2ME focuses on only the client (i.e., our servlets do not use J2ME technology), so we discuss J2ME after we discuss the application’s servlets. Until we discuss Section 12.4, we recommend that you regard the J2ME client as “just another client” that receives servlet-generated content (i.e., you need not know J2ME-related topics to understand servlet behavior). In Section 12.4, we discuss how the J2ME client receives and interprets this content—e.g., how the J2ME client presents this data in a user interface. We also recommend that you follow the instructions in Section 12.5 for installing and configuring the software in this case study.

### Fig. 12.2  Database contents of **tips.sql** (part 2 of 2).
12.2 WelcomeServlet Overview

We begin by discussing class **WelcomeServlet** (Fig. 12.1), which redirects a client request to a static page that displays Tip-Test game instructions—this static page contains a link to the **TipTestServlet**, which enables the user to play the game.

Clients interact with servlets by making get or post requests to the servlets. Clients send get requests to **WelcomeServlet** to get the “welcome screen.” When a client sends a get request to **WelcomeServlet**, method doGet (lines 15–39) handles the request.

Each client type receives a different welcome screen from the servlet, because each client type supports a different content type. For example, Internet Explorer receives **index.html** as a welcome screen, because Internet Explorer can render XHTML documents. On the other hand, the Openwave UP simulator receives **index.wml**, because a WAP browser can render only WML documents. The Sun MIDP-device emulator can render only plain text, so **WelcomeServlet** sends **index.txt** to this device. The Pixo browser for i-mode can render cHTML (compact-HTML), so the servlet sends a different **index.html** than the one for Internet Explorer.

```java
// WelcomeServlet.java
// Delivers appropriate "Welcome" screen to client
package com.deitel.advjhtp1.wireless;

// Java core package
import java.io.*;

// Java extension packages
import javax.servlet.*;'nimport javax.servlet.http.*;

public class WelcomeServlet extends HttpServlet {

    // respond to get request
    protected void doGet( HttpServletRequest request,
                          HttpServletResponse response )
        throws ServletException, IOException {

        // determine User-Agent header
        String userAgent = request.getHeader( "User-Agent" );

        // send welcome screen to appropriate client
        if ( userAgent.indexOf( ClientUserAgentHeaders.IE ) != -1 )
            sendIEClientResponse( request, response );

        else if ( userAgent.indexOf( ClientUserAgentHeaders.WAP ) != -1 )
            sendWAPClientResponse( request, response );

    }

}
```

Fig. 12.3 Class **WelcomeServlet** sends an introductory screen that provides game directions to a client (part 1 of 3).

---

1. At the time of this writing, J2ME clients can interpret XML documents only by using proprietary software—i.e., there does not exist a standard for J2ME clients to integrate XML.
else if ( userAgent.indexOf( // i-mode
    ClientUserAgentHeaders.IMODE ) != -1 )
    sendIModeClientResponse( request, response );

else if ( userAgent.indexOf( // J2ME
    ClientUserAgentHeaders.J2ME ) != -1 )
    sendJ2MEClientResponse( request, response );

} // end method doGet

// send welcome screen to IE client
private void sendIEClientResponse(
    HttpServletRequest request, HttpServletResponse response )
    throws IOException, ServletException
{
    redirect( "text/html", "/XHTML/index.html", request,
        response );
}

// send welcome screen to Nokia WAP client
private void sendWAPClientResponse(
    HttpServletRequest request, HttpServletResponse response )
    throws IOException, ServletException
{
    redirect( "text/vnd.wap.wml", "/WAP/index.wml", request,
        response );
}

// send welcome screen to i-mode client
private void sendIModeClientResponse(
    HttpServletRequest request, HttpServletResponse response )
    throws IOException, ServletException
{
    redirect( "text/html", "/iMode/index.html", request,
        response );
}

// send welcome screen to J2ME client
private void sendJ2MEClientResponse(
    HttpServletRequest request, HttpServletResponse response )
    throws IOException
{
    // send J2ME client text data
    response.setContentType( "text/plain" );
    PrintWriter out = response.getWriter();

    // open file to send J2ME client
    BufferedReader bufferedReader =
        new BufferedReader( new FileReader( 
            getServletContext().getRealPath( 
                "j2me/index.txt" ) ) );

Fig. 12.3  Class WelcomeServlet sends an introductory screen that provides game directions to a client (part 2 of 3).
Before responding to a client, method `doGet` must determine what type of client made the request. Each client includes a `User-Agent` header with each request. This header contains information on the type of client requesting data from the server. Interface `ClientUserAgentHeaders` (Fig. 12.4) lists a unique `User-Agent` header substring for each client in our application. For example, a `User-Agent` header for Microsoft Internet Explorer running on Windows 2000 might be

Mozilla/4.0 (compatible; MSIE 5.0; Windows NT 5.0)

We search for the substring "MSIE 5" in the `User-Agent` header to recognize Internet Explorer requests from different platforms. Also, `WelcomeServlet` will recognize variance among other versions of Internet Explorer 5 (e.g., v.5.0, v.5.5, etc.). For example, the `User-Agent` header for a Windows 98 client might not be identical to the one shown, but the header will contain the "MSIE 5" substring.

Line 20 of class `WelcomeServlet` extracts the `User-Agent` header from the `HttpServletRequest`. Lines 23–37 determine which client made the request by matching the `User-Agent` header with the ones in interface `ClientUserAgentHeaders`. If an Internet Explorer browser made the request, line 25 invokes method `sendIEClientResponse` (lines 42–48). Lines 46–47 calls method `redirect` (lines 96–108), which redirects the request to a static page. For Internet Explorer, line 101 calls
method `setContentType` of the `HttpServletResponse` object to set the MIME type to `text/html`—the MIME type for XHTML clients. The MIME type (`Multipurpose Internet Mail Extensions`) helps browsers determine how to interpret data. Lines 102–107 redirect the request to `index.html` by creating a `RequestDispatcher` object with the name of the static page and invoking method `forward` of the `RequestDispatcher`. The IE browser then displays `index.html` (Fig. 12.5).
If the Openwave UP simulator made the request, line 29 invokes method `sendWAP-ClientResponse` (lines 51–57), which invokes method `redirect` with the MIME type to `text/vnd.wap.wml`—the MIME type for WML clients—and redirects the request to `index.wml`. At this point, the simulator displays `index.wml`, as shown in Fig. 12.6.

If the Pixo i-mode browser made the request, line 33 invokes method `sendIMode-ClientResponse` (lines 60–66). This method also invokes method `redirect`, but sets the MIME type to `text/html`—the MIME type for cHTML clients—and redirects the request to a cHTML version of `index.html`. At this point, the Pixo browser displays `index.html`, as shown in Fig. 12.7.

If the Sun MIDP-device emulator made the request, line 37 invokes method `sendJ2MEClientResponse` (lines 69–93). To transfer `index.txt` to the J2ME client, the servlet must send `index.txt` through a stream between the servlet and the J2ME client. Line 74 sets the MIME type to `text/plain`. Line 75 invokes method `getWriter` of class `HttpServletRequest` to obtain a `PrintWriter` for sending data to the client. Lines 78–81 create a `BufferedReader` that reads `index.txt` from the `j2me` directory located in the servlet context. Lines 86–89 send each line from `BufferedReader` to the client. At this point, Sun’s emulator displays `index.txt`, as shown in Fig. 12.8.

---

![Fig. 12.6](image_url)

**Fig. 12.6** WelcomeServlet output (`index.wml`) for WAP client. (Image of UP.SDK courtesy Openwave Systems Inc. Openwave, the Openwave logo, and UP.SDK are trademarks of Openwave Systems Inc. All rights reserved.)
Fig. 12.7 *WelcomeServlet* output (**index.html**) for i-mode client. (Courtesy of Pixo, Inc.)

Fig. 12.8 *WelcomeServlet* output (**index.txt**) for J2ME client. (Courtesy of Sun Microsystems, Inc.)
12.3 TipTestServlet Overview

In this section, we discuss how the TipTestServlet (Fig. 12.9) generates and sends Tip-Test questions and answers to each client. Each welcome screen contains a link to an “instructions” screen—info.html, info.wml or info.txt, depending on the client—that provides game information. The instructions screen contains a link to TipTestServlet.\(^2\)

If the user accesses TipTestServlet and the servlet has not received a previous request from a client, the servlet container calls method init (lines 30–67) to initialize TipTestServlet. This method performs operations that TipTestServlet need only perform once, such as loading the JDBC driver, connecting to the database and instantiating objects for creating XML documents. Lines 36–38 retrieve the class name for the JDBC-database driver from the servlet-initialization parameter in web.xml (which we discussed when we installed Tomcat), and line 40 loads this driver.

```java
1 // TipTestServlet.java
2 // TipTestServlet sends Tip Test to clients.
3 package com.deitel.advjhtp1.wireless;
4
5 // Java core packages
6 import java.io.*;
7 import java.sql.*;
8 import java.util.*;
9
10 // Java extension packages
11 import javax.servlet.*;
12 import javax.servlet.http.*;
13 import javax.xml.parsers.*;
14 import javax.xml.transform.*;
15 import javax.xml.transform.dom.*;
16 import javax.xml.transform.stream.*;
17
18 // import third-party packages
19 import org.w3c.dom.*;
20 import org.xml.sax.SAXException;
21
22 public class TipTestServlet extends HttpServlet {
23    private Connection connection; // database connection
24
25    private DocumentBuilderFactory factory;
26    private TransformerFactory transformerFactory;
27
28    // initialize servlet
29    public void init() throws ServletException
30    {

Fig. 12.9 TipTestServlet handles game logic and sends Tip Test to clients (part 1 of 13).

2. The client is not required to access TipTestServlet through WelcomeServlet. The user can type the TipTestServlet URL in the browser to access TipTestServlet and bypass the “welcome screen” that WelcomeServlet generates.
// load database driver and instantiate XML factories
try {
    // get JDBC driver from servlet container
    String jdbcDriver =
        getServletConfig().getInitParameter(
            "JDBC_DRIVER" );
    Class.forName( jdbcDriver ); // load JDBC driver
    // get database URL from servlet container
    String databaseUrl =
        getServletConfig().getInitParameter(
            "DATABASE_URL" );
    connection = DriverManager.getConnection( databaseUrl );
    // create a Factory to build XML Documents
    factory = DocumentBuilderFactory.newInstance();
    // create new TransformerFactory
    transformerFactory = TransformerFactory.newInstance();
} // end try

// handle exception database driver class does not exist
catch ( ClassNotFoundException classNotFoundException ) {
    classNotFoundException.printStackTrace();
}

// handle exception in making Connection
catch ( SQLException sqlException ) {
    sqlException.printStackTrace();
}

} // end method init

// respond to get requests
protected void doGet( HttpServletRequest request,
                      HttpServletResponse response )
    throws ServletException, IOException
{
    // get Statement from database, then send Tip-Test Question
    try {
        // SQL query to database
        Statement statement = connection.createStatement();
        // get database information using SQL query
        ResultSet resultSet =
            statement.executeQuery( "SELECT * FROM tipInfo" );

Fig. 12.9  TipTestServlet handles game logic and sends Tip Test to clients (part 2 of 13).
// parse and send ResultSet to client
if ( resultSet != null ) {

// ensure that client does not cache questions
response.setHeader( "Cache-Control",
    "no-cache, must-revalidate" );
response.setHeader( "Pragma", "no-cache" );

sendTipTestQuestion( request, response, resultSet );
}

statement.close(); // close Statement
}

// handle exception in executing Statement
catch ( SQLException sqlException ) {
    sqlException.printStackTrace();
}

} // end method doGet

// respond to post requests
protected void doPost( HttpServletRequest request, HttpServletResponse response )
throws ServletException, IOException {

// send ResultSet to appropriate client
try {

    // determine User-Agent header
    String userAgent = request.getHeader( "User-Agent" );

    // if Internet Explorer is requesting client
    if ( userAgent.indexOf( ClientUserAgentHeaders.IE ) != -1 ) {
        Document document = createXMLTipTestAnswer( request );

        // set appropriate Content-Type for client
        response.setContentType( "text/html" );

        // send XML content to client after XSLT
        applyXSLT( "XHTML/XHTMLTipAnswer.xsl", document, response );

    }
    else if ( userAgent.indexOf( ClientUserAgentHeaders.WAP ) != -1 ) {

Fig. 12.9 TipTestServlet handles game logic and sends Tip Test to clients (part 3 of 13).
Document document =
    createXMLTipTestAnswer( request );

    // set appropriate Content-Type for client
response.setContentType( "text/vnd.wap.wml" );

    // send XML content to client after XSLT
applyXSLT( "WAP/WAPTipAnswer.xsl", document,
            response );
}

    // if i-mode client is requesting client
else if ( userAgent.indexOf(
        ClientUserAgentHeaders.IMODE ) != -1 ) {

    Document document =
        createXMLTipTestAnswer( request );

    // set appropriate Content-Type for client
response.setContentType( "text/html" );

    // send XML content to client after XSLT
applyXSLT( "iMode/IMODETipAnswer.xsl", document,
            response );
}

    // if J2ME client is requesting client
else if ( userAgent.indexOf(
        ClientUserAgentHeaders.J2ME ) != -1 )
    sendJ2MEAnswer( request, response );

    } // end try

    // handle exception if Document is null
    catch ( NullPointerException nullPointerException ) {
        nullPointerException.printStackTrace();
    }

} // end method doPost

    // send Tip-Test data to client
private void sendTipTestQuestion(
    HttpServletRequest request, HttpServletResponse response,
    ResultSet resultSet ) throws IOException {

    // send ResultSet to appropriate client
    try {

        // determine User-Agent header
        String userAgent = request.getHeader( "User-Agent" );
    }

Fig. 12.9  TipTestServlet handles game logic and sends Tip Test to clients (part 4 of 13).
// if Internet Explorer is requesting client
if ( userAgent.indexOf(
    ClientUserAgentHeaders.IE ) != -1 ) {

    Document document =
    createXMLTipTestQuestion( resultSet, request,
        request.getContextPath() + "/XHTML/images/",
        ".gif" );

    // set appropriate Content-Type for client
    response.setContentType( "text/html" );
    applyXSLT( "XHTML/XHTMLTipQuestion.xsl", document,
        response );
}

// if WAP client is requesting client
else if ( userAgent.indexOf(
    ClientUserAgentHeaders.WAP ) != -1 ) {

    Document document =
    createXMLTipTestQuestion( resultSet, request,
        request.getContextPath() + "/WAP/images/",
        ".wbmp" );

    // set appropriate Content-Type for client
    response.setContentType( "text/vnd.wap.wml" );
    applyXSLT( "WAP/WAPTipQuestion.xsl", document,
        response );
}

// if i-mode client is requesting client
else if ( userAgent.indexOf(
    ClientUserAgentHeaders.IMODE ) != -1 ) {

    Document document =
    createXMLTipTestQuestion( resultSet, request,
        request.getContextPath() + "/iMode/images/",
        ".gif" );

    // set appropriate Content-Type for client
    response.setContentType( "text/html" );
    applyXSLT( "iMode/IMODETipQuestion.xsl", document,
        response );
}

// if J2ME client is requesting client
else if ( userAgent.indexOf(
    ClientUserAgentHeaders.J2ME ) != -1 )
    sendJ2MEClientResponse( resultSet, request,
        response );

} // end try

Fig. 12.9  TipTestServlet  handles game logic and sends Tip Test to clients (part 5 of 13).
// handle exception if Document is null
catch ( NullPointerException nullPointerException ) {
    nullPointerException.printStackTrace();
}

} // end method sendTipTestQuestion

// send Tip Test to Internet Explorer client
private Document createXMLTipTestQuestion(
    ResultSet resultSet, HttpServletRequest request,
    String imagePrefix, String imageSuffix )
throws IOException
{
    // convert ResultSet to two-dimensional String array
    String resultTable[][] = getResultTable( resultSet );

    // create random-number generator
    Random random = new Random( System.currentTimeMillis() );

    // create 4 random tips
    int randomRow[] = getRandomIndices( random );

    // randomly determine correct index from 4 random indices
    int correctAnswer = Math.abs( random.nextInt() ) %
        randomRow.length;

    int correctRow = randomRow[ correctAnswer ];

    // open new session
    HttpSession session = request.getSession();

    // store correct answer in session
    session.setAttribute( "correctAnswer",
        new Integer( correctAnswer ) );

    // store correct tip name
    session.setAttribute( "correctTipName", new String( resultTable[ correctRow ][ 1 ] ) );

    // store correct tip description
    session.setAttribute( "correctTipDescription", new String( resultTable[ correctRow ][ 2 ] ) );

    // determine image to send client
    String imageName = imagePrefix +
        resultTable[ correctRow ][ 3 ] + imageSuffix;

    // create XML document based on randomly determined info
    try {
        
Fig. 12.9  TipTestServlet handles game logic and sends Tip Test to clients (part 6 of 13).
// create document
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.newDocument();

// create question root Element
Element root = document.createElement("question");
document.appendChild(root);

// append Element image, which references image name
Element image = document.createElement("image");
image.appendChild(document.createTextNode(imageName));
root.appendChild(image);

// create choices Element to hold 4 choice Elements
Element choices = document.createElement("choices");

// append 4 choice Elements that represent user choices
for (int i = 0; i < randomRow.length; i++) {
    // determine choice Elements from resultTable
    Element choice = document.createElement("choice");
    choice.appendChild(document.createTextNode(resultTable[randomRow[i]][4]));

    // set choice Element as correct or incorrect
    Attr attribute = document.createAttribute("correct");
    if (i == correctAnswer)
        attribute.setValue("true");
    else
        attribute.setValue("false");

    // append choice Element to choices Element
    choice.setAttributeNode(attribute);
    choices.appendChild(choice);
}
root.appendChild(choices);

return document;

} // end try

// handle exception building Document
catch (ParserConfigurationException parserException) {
parsingException.printStackTrace();
}

return null;

Fig. 12.9  TipTestServlet handles game logic and sends Tip Test to clients (part 7 of 13).
341 } // end method createXMLTipTestQuestion
342
343 // send tip test to J2ME client
344 private void sendJ2MEClientResponse( ResultSet resultSet,
345 HttpServletRequest request,
346 HttpServletResponse response ) throws IOException
347 {
348 // convert ResultSet to two-dimensional String array
349 String resultTable[][] = getResultTable( resultSet ];
350
351 // create random-number generator
352 Random random = new Random( System.currentTimeMillis() ];
353
354 // create 4 random tips
355 int randomRow[] = getRandomIndices( random ];
356
357 // randomly determine correct index from 4 random indices
358 int correctAnswer = Math.abs( random.nextInt( ) ) %
359 randomRow.length;
360
361 int correctRow = randomRow[ correctAnswer ];
362
363 // open old session
364 HttpSession session = request.getSession();
365
366 // store correct answer in session
367 session.setAttribute( "correctAnswer", new Integer( correctAnswer ] );
368
369 // store correct tip name in session
370 session.setAttribute( "correctTipName", new String( resultTable[ correctRow ][ 1 ] ] );
371
372 // store correct tip description in session
373 session.setAttribute( "correctTipDescription", new String( resultTable[ correctRow ][ 2 ] ] );
374
375 // send J2ME client image name
376 String imageName = "/j2me/images/" +
377 resultTable[ correctRow ][ 3 ] + ".png";
378
379 response.setContentType( "text/plain" ];
380 PrintWriter out = response.getWriter();
381 out.println( imageName );
382
383 // send J2ME client test
384 for ( int i = 0; i < randomRow.length; i++ )
385 out.println( resultTable[ randomRow[ i ] ][ 4 ] ] );
386
387 } // end method sendJ2MEClientResponse
388

Fig. 12.9 TipTestServlet handles game logic and sends Tip Test to clients (part 8 of 13).
private String[][] getResultTable( ResultSet resultSet )
{
    // create table of Strings to store ResultSet
    String resultTable[][] = new String[ 7 ][ 5 ];
    for ( int i = 0; i < 7; i++ ) {
        for ( int j = 0; j < 5; j++ )
            resultTable[ i ][ j ] = "";
    }
    // store all columns in table
    try {
        // for each row in resultSet
        for ( int row = 0; resultSet.next(); row++ ) {
            // for each column in resultSet
            for ( int column = 0; column < 5; column++ ) {
                // store resultSet element in resultTable
                resultTable[ row ][ column ] +=
                    resultSet.getObject( column + 1 );
            }
        }
        // handle exception if servlet cannot get ResultSet Object
        catch ( SQLException sqlException ) {
            sqlException.printStackTrace();
            return null;
        }
        return resultTable;
    } // end method getResultTable

    // get 4 randomly generated indices from resultTable
    private int[] getRandomIndices( Random random )
    {
        // create list containing row indices for resultTable
        int list[] = new int[ 7 ];
        for ( int i = 0; i < list.length; i++ )
            list[ i ] = i;
        int randomRow[] = new int[ 4 ];
        // select 4 randomly generated indices from list
        for ( int i = 0; i < randomRow.length; i++ )
            randomRow[ i ] = getRandomRow( list, random );

    }

} // end method
private int getRandomRow( int list[], Random random ) {
    // get random element from list
    int randomRow = Math.abs( random.nextInt() ) % list.length;
    while ( list[ randomRow ] < 0 )
        randomRow = Math.abs( random.nextInt() ) % list.length;
    list[ randomRow ] = -1; // nullify element
    return randomRow;
} // end method getRandomRow

// apply XSLT style sheet to XML document
private void applyXSLT( String xslFile,
    Document xmlDocument, HttpServletResponse response )
    throws IOException
{
    // apply XSLT
    try {
        // open InputStream for XSL document
        InputStream xslStream =
            getServletContext().getResourceAsStream( xslFile );
        // create StreamSource for XSLT document
        Source xslSource = new StreamSource( xslStream );
        // create DOMSource for source XML document
        Source xmlSource = new DOMSource( xmlDocument );
        // get PrintWriter for writing data to client
        PrintWriter output = response.getWriter();
        // create StreamResult for transformation result
        Result result = new StreamResult( output );
        // create Transformer for XSL transformation
        Transformer transformer =
            transformerFactory.newTransformer( xslSource );
        // transform and deliver content to client
        transformer.transform( xmlSource, result );
    } // end try

TipTestServlet handles game logic and sends Tip Test to clients (part 10 of 13).
```java
    // handle exception transforming content
    catch ( TransformerException exception ) {
        exception.printStackTrace();
    }
}

// create XML Document that stores Tip Test answer
private Document createXMLTipTestAnswer(
    HttpServletRequest request ) throws IOException {
    // get session
    HttpSession session = request.getSession();

    // match correct answer with session answer
    Integer integer =
        ( Integer ) session.getAttribute( "correctAnswer" );
    int correctAnswer = integer.intValue();

    // give client correct tip name and description
    String correctTipName =
        ( String ) session.getAttribute( "correctTipName" );
    String correctTipDescription =
        ( String ) session.getAttribute( "correctTipDescription" );

    // get user selection
    int selection = Integer.parseInt(
        request.getParameter( "userAnswer" ) );
    String answer;

    // determine if user answer is correct
    if ( correctAnswer == selection )
        answer = "Correct";
    else
        answer = "Incorrect";

    // get link to TipTestServlet
    String servletName = request.getContextPath() + "/" +
    getServletConfig().getServletName();

    // create XML document based on randomly determined info
    try {
        // create document
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();
    }
```

**Fig. 12.9** TipTestServlet handles game logic and sends Tip Test to clients (part 11 of 13).
// create question root Element
Element root = document.createElement("answer");
document.appendChild(root);

// append Element that informs client of correct answer
Element correct = document.createElement("correct");
correct.appendChild(document.createTextNode(answer));
root.appendChild(correct);

// append Element that describes tip name
Element name =
document.createElement("correctTipName");
name.appendChild(document.createTextNode(correctTipName));
root.appendChild(name);

// append Element that describes tip description
Element description =
document createElement("correctTipDescription");
description.appendChild(document.createTextNode(correctTipDescription));
root.appendChild(description);

// append Element that links to TipTestServlet
Element servletLink =
document.createElement("servletName");
servletLink.appendChild(document.createTextNode(servletName));
root.appendChild(servletLink);

return document;
}

// handle exception building Document
catch (ParserConfigurationException parserException) {
    parserException.printStackTrace();
}

return null;

} // end method createXMLTipTestAnswer

// send answer to J2ME client
private void sendJ2MEAnswer(HttpServletRequest request,
HttpServletResponse response) throws IOException {

    // get client test response
    BufferedReader in = request.getReader();
    int selection = Integer.parseInt(in.readLine().trim());

Fig. 12.9 TipTestServlet handles game logic and sends Tip Test to clients (part 12 of 13).
Using method `getInitParameter` in line 37 allows us to specify the database outside the servlet—we can change the database (e.g., change from Cloudscape to mySQL) by
modifying the `<param-value>` element in the `<init-param>` element in `web.xml`, without having to recompile the servlet. For this case study, this element contains

```xml
<init-param>
    <param-name>JDBC_DRIVER</param-name>
    <param-value>
        COM.cloudscape.core.RmiJdbcDriver
    </param-value>
</init-param>
```

Lines 43–45 retrieve the database URL from `web.xml`, and line 47 makes the connection to this URL—note that method `destroy` (lines 632–644) closes this connection. If we decide to change the location of the database, we need only to modify the `<param-value>` element in a separate `<init-param>` element in `web.xml`. For this case study, this element contains

```xml
<init-param>
    <param-name>DATABASE_URL</param-name>
    <param-value>jdbc:cloudscape:rmi:tips</param-value>
</init-param>
```

Lines 50–53 create objects that we use later to build XML Documents and use Transformers to apply XSLTs. We discuss XML-related Java packages later in this section.

`TipTestServlet` calls method `doGet` (lines 70–103) to handle the get request. Line 78 creates a `Statement` from the `Connection` instantiated in method `init`. Lines 81–82 obtain a `ResultSet` of all database elements from this `Statement`. As discussed in the introduction, our database (the `ResultSet`) contains seven rows and five columns—resulting in 35 `String` objects. Lines 88–90 set the `HttpServletResponse`’s header so the client will not cache the information that `TipTestServlet` sends. Line 92 calls method `sendTipTestQuestion` (lines 176–244) to perform the game logic and send the client the Tip-Test question. Line 184 extracts the `User-Agent` header from the `HttpServletRequest`. Lines 187–235 determine which client made the request by matching the `User-Agent` header with those in interface `ClientUserAgentHeaders`. This client-determination process is identical to the one in `WelcomeServlet`.

Until this point in execution, the servlet has behaved identically for all client requests. Now, `TipTestServlet` performs different operations based on the type of client making the request. These operations are almost identical among the Internet Explorer, Openwave UP simulator and the Pixo i-mode browser. The servlet’s operations for the J2ME client differ from that of the other three clients, because the J2ME client cannot interpret XML documents without using non-standard software (although this is likely to change in the future). We now trace the servlet’s behavior for each client type’s request. We begin discussing `TipTestServlet`’s behavior for an Internet Explorer request. Next, we discuss the servlet’s behavior for a Openwave UP simulator request. We then discuss the servlet’s behavior for a Pixo i-mode browser request. Finally, we discuss the servlet’s behavior for a Sun MIDP-device client request.

### 12.3.1 Internet Explorer request

If Internet Explorer sent the request, lines 190–193 invoke method `createXMLTipTestQuestion` (lines 247–341), which creates and returns an XML `Document` that
contains the Tip-Test question. Interface Document of package org.w3c.dom represents an XML document’s top-level node, which provides access to all the document’s nodes. Line 253 calls method getResultSet (lines 393–428), which converts the ResultSet into a two-dimensional array of Strings. We use this method so we have an easier time accessing individual elements of the ResultSet through an array. Lines 396–402 instantiate this array, and lines 408–417 transfer the ResultSet content to the array.

When method getResultSet returns the String array, line 256 creates a Random object that enables TipTestServlet to select the tip image and possible answers randomly. Line 259 passes a reference to the Random object to method getRandomeRandomIndices (lines 431–447), which returns an array of four integers. Each integer is a distinct, random index that corresponds to a tip in the two-dimensional String array that method getResultSet generated. Lines 434–437 create an array of seven integers called list—each integer represents that integer’s index in the array (i.e., list[0] = 0, list[1] = 1, etc.). Line 439 creates an empty array of four integers that correspond to the four possible tip answers. Lines 442–443 invoke method getRandomRow (lines 450–462) for each element in the four-integer array. Line 453 select an integer at random from the seven-integer array. If the integer has been selected previously—indicated by a -1 value—lines 455–456 select an integer that has not been selected. If the integer has not been selected previously, line 458 nullifies this array element by setting that element to -1. Nullifying this element ensures that method getRandomRow will not return duplicate integers. Line 460 returns the integer.

Array randomRow contains four distinct integers that correspond to tips in resultTable. Lines 262–263 randomly select an integer from randomRow. This integer indicates the “correct” answer in the game—specifically, this integer represents the resultTable row that holds the information for the correct tip (line 265). TipTestServlet stores this information in an HttpSession object (line 268), so when the client sends the user’s selection back to TipTestServlet, TipTestServlet can check this selection against the correct answer stored in the HttpSession object. An HttpSession object acts like a cookie—both can store name/value pairs. In session terminology, these pairs are called attributes. We may store attributes in an HttpSession object with method setAttribute. Lines 271–280 store the correct answer, tip name and tip description in the HttpSession object. Using the correct answer and parameters imagePrefix and imageSuffix of method createXMLTipTestQuestion, lines 283–284 determine the proper image to send to the client.

At this point, TipTestServlet has determined the Tip-Test question (the image and four possible answers) to send the client. We now explain how TipTestServlet marks up this data as an XML Document, applies an XSL transformation to this Document, then sends the resulting document to the client.


Class Element represents an element node in an XML document. A Document creates an Element with method createElement. Lines 294–295 create Element question and assign it as the root Element in the Document. Lines 298–301 create Element image from the correct tip’s image and assign Element image as a child of
Element question. Line 304 creates Element choices, which hold the four choice Elements created on lines 307–326. Each choice Element represents a possible answer for the question. Lines 315–324 include an attribute that specifies whether the choice is correct or incorrect—obviously, only one attribute specifies a value of correct. Line 325 appends the choice Elements to Element choices, and line 328 appends choices to root Element question.

When method createXMLTipTestQuestion returns the XML Document, TipTestServlet must apply an XSLT to this Document and send the result of the transformation to the client. Line 124 sets the HttpServletResponse’s MIME type to text/html, because the XSL transformation generates an XHTML document. Lines 127–128 call method applyXSLT (465–502) to apply XHTMLTipQuestion.xsl (Fig. 12.10) to the XML Document. Lines 473–474 open an InputStream for XHTMLTipQuestion.xsl. Line 477 creates a StreamSource from this InputStream for the XSL document. Line 480 creates a DOMSource for the XML Document. Line 486 creates a Result for the resulting transformation—in this case, an XHTML document. TipTestServlet uses a PrintWriter to create the Result, so TipTestServlet will send the resulting transformation directly to the client. Lines 489–490 create a Transformer using the XSL StreamSource and the TransformerFactory (instantiated in method init). The Transformer is the object that applies the XSLT to the XML document. Line 493 invokes method transform of class Transformer to apply the transformation to produce the XHTML document.

In Fig. 12.10, lines 9–11 specify the DTD and other output options using the xsl:output element. Line 15 specifies that Element question is the root Element in the XML Document that XHTMLTipQuestion.xsl will transform. Line 16 begins the XHTML document with the element html. Lines 22–86 contain the body element, which contains the tip image and the four possible answers. Lines 27–29 display the image associated with the image Element in the XML Document. Lines 40–83 create a form with four possible answers. Lines 47–77 create four radio buttons associated with these answers. Line 82 creates a Submit button so the user can submit the selection. Figure 12.11 shows Tip Test in action after the servlet applies the XSLT to the XML Document. The image on the left shows the user selecting the answer, and the image on the right shows the user about to submit the answer.
Fig. 12.10 XHTMLTipQuestion.xsl transforms XML Tip-Test question to XHTML document (part 2 of 3).
Each radio button in the XHTML document contains a unique value, so when the user presses the Submit button, Internet Explorer sends the selected value to TipTest-Servlet (as post data). TipTestServlet (Fig. 12.9) invokes method doPost.
(lines 106–173) upon receiving post data. Lines 117–129 recognize that the request is from the Internet Explorer and call method createXMLTipTestAnswer (lines 505–589) to determine if the user’s answer is correct. Line 509 retrieves the HttpSession, and lines 512–522 extract the correct tip answer, name and description from the HttpSession. Lines 525–526 retrieve the user’s selection from the HttpServletRequest. Lines 531–534 match the user’s selection with the tip answer to determine if the user is correct. Lines 537–538 determine the TipTestServlet’s URL so clients can reconnect to TipTestServlet to receive another Tip-Test question.

Lines 544–576 mark up the HttpSession contents and TipTestServlet’s URL as an XML Document from the DocumentBuilder. Line 544 creates a DocumentBuilder, and line 545 creates an XML Document. Lines 548–549 create Element answer and assign it as the root Element in the Document. Lines 552–554 create Element correct, which stores whether the user is correct or incorrect. Line 555 assigns Element correct as a child of Element answer. Lines 558–561 and lines 565–568 create Elements correctTipName and correctTipDescription, which hold the name and description of the correct tip, respectively. Lines 562 and 569 assign these Elements as children of Element answer. Lines 572–575 create Element servletName, which holds TipTestServlet’s URL. Line 576 assigns this Element as a child of Element answer.

When method createXMLTipTestAnswer returns the XML Document (line 578), TipTestServlet must apply an XSLT to this Document and send the resulting transformation to the client. Line 124 sets the MIME type to text/html, and lines 127–128 call method applyXSLT to apply XHTMLTipAnswer.xsl (Fig. 12.12) to the XML Document.

```xml
<?xml version="1.0"?>
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:output method = "xml" omit-xml-declaration = "no"
    doctype-system = "DTD/xhtml1-strict.dtd"
    doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
  <!-- specify the root of the XML document -->
  <!-- that references this stylesheet -->
  <xsl:template match = "answer">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title>Tip Test Answer</title>
      </head>

Fig. 12.12 XHTMLTipAnswer.xsl transforms XML Tip-Test answer to XHTML document (part 1 of 2).
In Fig. 12.12, lines 9–11 specify the DTD using the `xsl:output` element. Line 15 specifies that `Element answer` is the root `Element` in the XML `Document` that `XHTMLTipAnswer.xsl` will transform. Line 16 begins the XHTML document. Line 26 uses `Element correct` from the XML `Document` to display whether the user’s selection is the correct answer. Lines 36 and 47 use `Elements correctTipName` and `correctTipDescription` to display the name and description of the correct tip, respectively. Line 53 uses `Element servletName` to provide a link to `TipTest-Servlet` so the user can receive another Tip-Test question. Figure 12.21 shows the Tip-Test answer in Internet Explorer.
12.3.2 WAP request

If the Openwave UP simulator made the original get request to TipTestServlet (Fig. 12.9), lines 202–214 determine that the requesting client is a WAP client. Lines 205–208 invoke method createXMLTipTestQuestion to create an XML Document that contains the Tip-Test question. Using the arguments for this method, we specify that the tip images are in wbmp format and are located in the WAP/images directory of the servlet-context path. Line 211 sets the MIME type to text/vnd.wap.wml to produce WML content. Lines 212–213 call method applyXSLT to apply WAPTtipQuestion.xsl (Fig. 12.14) to the XML Document.

```
<?xml version="1.0"?>
<!-- WAPTtipQuestion.xsl -->
<!-- WAP stylesheet -->
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
<xsl:output method = "xml" omit-xml-declaration = "no"
    doctype-system = "http://www.wapforum.org/DTD/wml_1.1.xml"
    doctype-public = "-//WAPFORUM//DTD WML 1.1//EN"/>
```

Fig. 12.13 Internet Explorer Tip-Test answer output screen.

Fig. 12.14 WAPTtipQuestion.xsl transforms XML Tip-Test question to WML document (part 1 of 2).
Chapter 12 Java-Based Wireless Applications Development and J2ME

Fig. 12.14 WAPTipQuestion.xsl transforms XML Tip-Test question to WML document (part 2 of 2).
In Fig. 12.14, lines 10–12 specify the DTD using the xsl:output element. Line 16 specifies that Element question is the root Element in the XML Document that WAPTipQuestion.xsl will transform. Line 18 begins the WML document with the wml element. Lines 19–30 declare the first card—or the page that displays WML content—that the browser will display. Lines 26–27 display the image associated with the image Element in the XML Document. The do element (lines 21–23) informs the simulator to show the second card when the user presses the OK button on the simulator. Lines 32–61 declare the second card onto which the UP simulator places the four possible answers. Using the choice Elements in the choices Element, lines 45–59 create a selection list that contains the possible answers. The do element (lines 33–38) informs the simulator to send the user’s selection to TipTestServlet when the user presses the Submit button.

Figure 12.15 shows Tip Test after TipTestServlet applies the XSLT to the XML Document. The image on the left shows the tip image, and the image on the right shows the user selecting the answer.
Each selection list item contains a unique value. When the user presses `Submit`, the UP simulator `posts` the selected value to `TipTestServlet`. `TipTestServlet` (Fig. 12.9) invokes method `doPost` when receiving `post` data. Lines 132–144 determine that the request is from the Openwave UP simulator. Lines 135–136 call method `createXMLTipTestAnswer` to create an XML `Document` that stores whether the user is correct, the name and description of the correct tip and `TipTestServlet`’s URL. Line 139 sets the MIME type to `text/vnd.wap.wml` to produce WML content and lines 157–158 call `applyXSLT` to apply `WAPTipAnswer.xsl` (Fig. 12.16) to the XML `Document`.

In Fig. 12.16, lines 10–12 specify the DTD using the `xsl:output` element. Line 16 specifies that `Element answer` is the root `Element` in the XML `Document` that `WAPTipAnswer.xsl` will transform. Line 18 begins the WML document with the `wml` element. Lines 20–48 declare the card (the answer screen) in this WML document. Line 29 uses `Element correct` from the XML `Document` to display whether the user’s selection is the correct answer. Lines 37 and 45 use `Elements correctTipName` and `correctTipDescription` to display the name and description of the correct tip, respectively. Using `Element servletName` in the XML `Document`, the `do` element (lines 22–26) provides a link to `TipTestServlet`, so the user can receive another Tip-Test question. Figure 12.17 shows the Tip-Test answer in the Openwave UP simulator.

```
<?xml version="1.0"?>
<!-- WAPTipAnswer.xsl -->
<!-- WAP stylesheet -->
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
  <!-- specify the root of the XML document -->
  <!-- that references this stylesheet -->
  <xsl:template match = "answer">
    <wml>
      <card id = "card1" title = "Tip Test Answer">
        <do type = "accept" label = "OK">
          <go method = "get"
            href = "/advjhtp1/tiptest"/>
        </do>
      </p>
    </xsl:template>
</xsl:stylesheet>
```

Fig. 12.16 WAPTipAnswer.xsl transforms answer to WML document (part 1 of 2).
Fig. 12.16 WAPTipAnswer.xsl transforms answer to WML document (part 2 of 2).

Fig. 12.17 Openwave UP simulator Tip-Test answer screen. (Image of UP.SDK courtesy Openwave Systems Inc. Openwave, the Openwave logo, and UP.SDK are trademarks of Openwave Systems Inc. All rights reserved.)
12.3.3 Pixo i-mode request

If the Pixo browser made the original get request to TipTestServlet (Fig. 12.9), lines 217–229 determine that the requesting client is an i-mode client. Lines 220–223 invoke method createXMLTipTestQuestion to create the XML Document that contains the Tip Test question. Using the arguments for this method, we specify that the tip images are in gif format and are located in the iMode/images directory of the servlet-context path. Line 226 sets the MIME type to text/html to produce cHTML content. Lines 227–228 call method applyXSLT to apply IMODETipQuestion.xsl (Fig. 12.18) to the XML Document.

```
<?xml version="1.0"?>
<!-- IMODETipQuestion.xsl -->
<!-- i-mode stylesheet -->
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:output method = "html" omit-xml-declaration = "yes"
doctype-public = "-//W3C//DTD Compact HTML 1.0 Draft//EN"/>
<xsl:template match = "question">
<html>
<head>
<title>Tip Test</title>
</head>
<body>
<p>
<!-- display image -->
<img name = "image" height = "40"
width = "40" alt = "Tip Image"
src = "{image}">
</p>
<p>
What is the name of the icon shown?
</p>
<form method = "post"
action = "/advjhtp1/tiptest">
</form>
</body>
</html>
```

Fig. 12.18 IMODETipQuestion.xsl transforms XML Tip-Test question to cHTML document (part 1 of 2).
In Fig. 12.18, lines 9–10 specify the DTD using the `<xsl:output>` element. Line 14 specifies that `Element question` is the root `Element` in the XML `Document` that
**IMODETipQuestion.xsl** will transform. Line 15 begins the cHTML document with the `html` element. Lines 21–85 contain the `body` element that stores the tip image and the four possible answers. Lines 26–29 display the image associated with the image `Element` in the XML `Document`. Lines 39–82 create a form for the four possible answers. Lines 46–76 create four radio buttons associated with these answers. Line 81 creates a `Submit` button so the user can submit the choice.

Figure 12.19 shows Tip Test in action after `TipTestServlet` applies the XSLT to the XML `Document`. The image on the left shows the user selecting the answer, and the image on the right shows the user about to submit the answer.

Each radio button in the cHTML document contains a unique value, so when the user presses `Submit`, the Pixo browser posts the selected value to `TipTestServlet`. `TipTestServlet` invokes method `doPost` upon receiving `post` data. Lines 147–159 determine that the request is from the Pixo browser. Lines 150–151 call method `createXMLLTipTestAnswer` to create an XML `Document` that stores whether the user is correct, the name and description of the correct tip, and `TipTestServlet`’s URL. Line 154 sets the MIME type to `text/html` to produce cHTML content, and lines 157–158 call method `applyXSLT` to apply `IMODETipAnswer.xsl` (Fig. 12.20) to the XML `Document`.

In Fig. 12.20, lines 9–10 specify the DTD using the `xsl:output` element. Line 14 specifies that `Element answer` is the root `Element` in the XML `Document` that `IMODETipAnswer.xsl` will transform. Line 15 begins the cHTML document with the `html` element. Line 25 uses `Element correct` from the XML `Document` to display whether the user’s selection is correct. Lines 35 and 45–46 use `Elements correctTipName` and `correctTipDescription` to display the name and description of the correct tip, respectively. Using `Element servletName`, line 52 provides a link to `TipTestServlet`, so the user can receive another Tip-Test question. Figure 12.21 shows the Tip-Test answer on the Pixo i-mode browser.
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" omit-xml-declaration = "yes"
doctype-public = "-//W3C//DTD Compact HTML 1.0 Draft//EN"/>
  <!-- specify the root of the XML document -->
  <!-- that references this stylesheet -->
  <xsl:template match = "answer">
    <html>
      <head>
        <title>Tip Test Answer</title>
      </head>
      <body>
        <p>
          <h1>
            <xsl:value-of select = "correct"/>
          </h1>
        </p>
        <p>
          <h2>Tip Name</h2>
        </p>
        <p>
          <h3>
            <xsl:value-of select = "correctTipName"/>
          </h3>
        </p>
        <p>
          <h2>Tip Description</h2>
        </p>
        <p>
          <h3>
            <xsl:value-of select = "correctTipDescription"/>
          </h3>
        </p>
        <p>
          <a href="{servletName}">Next Tip</a>
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>

Fig. 12.20 IMODETipAnswer.xsl transforms XML Tip-Test answer to cHTML document.
12.3.4 J2ME client request

TipTestServlet behaves differently if a J2ME client made the get request. TipTestServlet does not use XML for the J2ME client, because currently, a J2ME client cannot interpret XML data without using proprietary software. Sun Microsystems claims that “it is not yet clear how far in the future this Java-XML nirvana is, but we can say that WAP 2.0 will use XHTML as its markup… We will see thousands, then millions, of devices using XHTML in consumers’ hands in the not too distant future.”

We do not use XML for the J2ME client, so TipTestServlet does not send marked up data to our J2ME client. We could have used an XSLT to generate the plain text, but we chose not to for two reasons. Firstly, plain text is not well-formed and defeats the purpose of using XSLT, which should create well-formed documents. Secondly, the ResultSet in TipTestServlet already contains plain text, so converting the text to an XML Document, then converting the Document back to text, is convoluted.

In TipTestServlet (Fig. 12.9), lines 232–235 determine that a J2ME client made the request and invoke method sendJ2MEClientResponse (lines 344–390). Line 349 calls method getResultTable to convert the ResultSet into a two-dimensional array of Strings—resultTable, so we have instant access to database contents. Line 352 creates a Random object, which allows TipTestServlet to generate randomly the tip image and tip questions to display. Line 355 passes the Random object to method getRandomIndices, which returns an array of four integers—each integer is a distinct, randomly generated index that corresponds to a tip in resultTable.

Lines 358–359 randomly select an integer from randomRow—this integer is the “correct” answer in the game—specifically, this integer represents the row in resultTable that holds the information for the correct tip. Using the correct answer, line 361 determines the row in resultTable that contains the name and description of the correct tip. TipTestServlet stores this information in an HttpSession object (line 364), so when the J2ME client sends the user’s selection back to the server, TipTestServlet can check the user’s selection against the correct answer stored in the HttpSession object. Lines 367–376 store the correct answer and the tip name and description in the HttpSession object. Using the correct answer, lines 379–380 determine the proper image to send to the J2ME client. Line 382 sets the MIME type to text/plain, because J2ME clients interprets plain text. Line 383 creates a PrintWriter object through which TipTestServlet sends the data to the J2ME client. Line 384 sends the name of the tip image, and lines 387–388 send the four tip abbreviations to the client. Section 12.4.3 discusses in detail how the J2ME client displays the Tip Test and exchanges data between itself and TipTestServlet. Figure 12.22 shows a sample output after TipTestServlet sends the Tip-Test question to the J2ME client. The image on the left shows the tip image, and the image on the right shows the user selecting an answer.

When the client sends the user’s selection, TipTestServlet invokes method doPost. Lines 162–164 determine that the request is from the J2ME client and call method sendJ2MEAnswer (lines 592–629). Lines 596–597 use a BufferedReader object to read the user’s selection from the client. Line 600 sets the MIME type to text/plain, and line 601 gets a PrintWriter object through which TipTestServlet sends content. Line 604 retrieves the HttpSession that stores the correct tip answer, name and description—lines 607–617 extract this information from this HttpSession. Lines 620–623 match the user’s selection with the correct tip answer and determine if the user is correct. Using the PrintWriter object, lines 626–627 send the correct tip name and description to the client. Figure 12.23 shows the answer screen on the Sun MIDP-device emulator. The picture on the left shows the screen that contains the correct tip name and description. When the user scrolls down on this screen, the user will see the picture on the right, which shows the remainder of the tip description.
Chapter 12  Java-Based Wireless Applications Development and J2ME

12.4 Java 2 Micro Edition

This section introduces *Java 2 Micro Edition (J2ME™)*—a platform that enables developers to write applications for various consumer devices, such as set-top boxes, Web termi-
nals, embedded systems, mobile phones and cell pagers. We discuss the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP), which offer developers a set of APIs to write J2ME applications—called MIDlets—and deploy them across several mobile devices. We then discuss the lifecycle of a MIDlet and explain how we use J2ME in our case study.

12.4.1 Connected Limited Device Configuration (CLDC)

The Connected Limited Device Configuration (CLDC) is a set of APIs that allow developers to create applications for devices that have limited resources—i.e., limited screen size, memory, power and bandwidth. The J2ME CLDC contains both a virtual machine—an interpreter that runs applications—and a set of classes that developers can use to develop and run programs on resource-limited devices. The KVM—the virtual machine offered by the CLDC—runs J2ME applications (as the JVM runs J2SE applications). The “K” in KVM represents the word “kilo,” because J2ME applications are small enough to be measured in kilobytes.

The J2ME CLDC contains packages java.io, java.lang, java.util that developers use to perform such common operations as creating primitive data types, using simple data structures and sending and receiving data from networks. These packages are subsets of the J2SE packages java.io, java.lang and java.util—that is, the J2ME CLDC packages do not contain every class from the J2SE packages. Figure 12.24 lists the J2ME java.io, java.lang and java.util packages. To conserve space, we did not include exception or error classes. For the complete J2ME CLDC class list, visit java.sun.com/j2me/docs/pdf/cldcapi.pdf.

<table>
<thead>
<tr>
<th>Classes</th>
<th>java.io</th>
<th>java.lang</th>
<th>java.util</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>DataInput</td>
<td>Runnable</td>
<td>Enumeration</td>
</tr>
<tr>
<td></td>
<td>DataOutput</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes</td>
<td>ByteArrayInputStream</td>
<td>Boolean</td>
<td>Calendar</td>
</tr>
<tr>
<td></td>
<td>ByteArrayInputStream</td>
<td>Byte</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>DataInputStream</td>
<td>Character</td>
<td>HashTable</td>
</tr>
<tr>
<td></td>
<td>DataOutputStream</td>
<td>Class</td>
<td>Random</td>
</tr>
<tr>
<td></td>
<td>PrintStream</td>
<td>Integer</td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>InputStreamReader</td>
<td>Long</td>
<td>Timer</td>
</tr>
<tr>
<td></td>
<td>OutputStreamReader</td>
<td>Math</td>
<td>TimerTask</td>
</tr>
<tr>
<td></td>
<td>PrintStream</td>
<td>Object</td>
<td>TimeZone</td>
</tr>
<tr>
<td></td>
<td>OutputStream</td>
<td>Runtime</td>
<td>Vector</td>
</tr>
<tr>
<td></td>
<td>PrintStream</td>
<td>Short</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reader</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Writer</td>
<td>StringBuffer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>System</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thread</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Throwable</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 12.24 J2ME java.io, java.lang and java.util packages.
Chapter 12 Java-Based Wireless Applications Development and J2ME

Common Programming Error 12.1

Attempting to use J2SE packages in the KVM will result in a compilation error—the KVM cannot handle the volume of classes due to KVM’s limited resources.

One challenge of J2ME programming is that the API does not contain certain data types and classes that developers often “take for granted” in other Java platforms. For example, J2ME does not include floating-point operations, serializable objects, thread groups or JNI (Java Native Interface). As wireless-device technology advances, it is possible that future versions of J2ME will support those features.

12.4.2 Mobile Information Device Profile (MIDP)

The Mobile Information Device Profile (MIDP) is a set of APIs that allow developers to handle mobile-device-specific issues, such as creating user interfaces, permitting local storage and defining the lifecycles of MIDP client applications (MIDlets). Devices that run applications using the MIDP are called MIDP devices. Such devices include cell phones or pagers.

MIDP contains packages javax.microedition.lcdui, javax.microedition.io, javax.microedition.rms and javax.microedition.midlet. Package javax.microedition.lcdui contains classes that allow developers to construct user-interfaces for MIDlets, package javax.microedition.io enables networking between MIDlets and other systems, package javax.microedition.rms contains classes that permit local storage, and package javax.microedition.midlet defines the MIDlet lifecycle. Figure 12.26 lists the MIDP javax.microedition.lcdui and javax.microedition.io packages. Figure 12.26 lists the MIDP javax.microedition.rms and javax.microedition.midlet packages. To conserve space, we did not include exception or error classes. For the complete J2ME MIDP class list, visit

java.sun.com/products/midp/midp-wirelessapps-wp.pdf

<table>
<thead>
<tr>
<th>Classes</th>
<th>javax.microedition.lcdui</th>
<th>javax.microedition.io</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>Choice</td>
<td>Connection</td>
</tr>
<tr>
<td></td>
<td>CommandListener</td>
<td>ContentConnection</td>
</tr>
<tr>
<td></td>
<td>ItemListener</td>
<td>Datagram</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DatagramConnection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HttpResponse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputConnection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OutputConnection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StreamConnection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>StreamConnectionNotifier</td>
</tr>
</tbody>
</table>
To run a MIDP application, a MIDP device requires a monochrome display of at least 96 pixels x 54 pixels, a two-way wireless network, some input device (such as a one-handed keypad or touch screen), at least 128 kilobytes for CLDC/MIDP classes and at least 32 kilobytes for the KVM. A MIDlet will run on any device that meets these requirements.
12.4.3 TipTestMIDlet Overview

In Section 12.3.4, we discussed how a servlet sends data to a J2ME client called a MIDlet. We discussed that MIDlets cannot interpret XML documents without proprietary software packages and must receive all information through streams or PrintReader objects. This section defines a MIDlet more specifically, discusses the MIDlet lifecycle and introduces TipTestMIDlet—the MIDlet in our case study.

A MIDlet is a Mobile Information Device application that runs on a MIDP device. The name is similar to the terms “applet” and “servlet,” because these applications share similar characteristics—for example, each has a lifecycle and occupies various states during program execution. Also, the developer does not invoke a constructor for objects of these classes (Applet, HttpServlet and MIDlet) explicitly to instantiate these objects. We discussed in Section 9.2.1 that a servlet container loads the servlet into memory—normally in response to the first request that servlet receives. MIDlets are loaded in a similar manner. MIDP developers store several MIDlets in a jar file—called a MIDlet suite—on a server. The MIDP device contains a program called the application management software (AMS), which downloads the MIDlet suite from the server, opens the MIDlet suite, then launches the user-specified MIDlet on the MIDP device.

The AMS uses an application descriptor file to load the MIDlet application. This file, which has a .jad extension, contains information such as the MIDlets in the MIDlet suite, the MIDlet suite’s size and URL, each MIDlet’s name, vendor and version, and the MIDP device’s profile and configuration. The AMS uses this information to ensure that the MIDlet application will run on the given MIDP device. Both the J2ME Wireless Toolkit and Forte create this file when creating a new MIDlet suite (see Section 12.5). The code below shows the structure of TipTestMIDlet’s application descriptor file.

```
MIDlet-1: TipTestMIDlet, TipTestMIDlet.png,
        com.deitel.advjhtpl.wireless.TipTestMIDlet
MIDlet-Jar-Size: 9577
MIDlet-Jar-URL: TipTestMIDlet.jar
MIDlet-Name: TipTestMIDlet
MIDlet-Vendor: Sun Microsystems
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

Figure 12.27 shows the TipTestMIDlet. Before we discuss how TipTestMIDlet retrieves information from the server, we must discuss the lifecycle of a MIDlet. Every MIDlet must extend class MIDlet (line 13) from package javax.microedition.midlet (line 9). The lifecycle begins when the AMS calls the MIDlet’s constructor (lines 43–75) to launch the MIDlet. The MIDlet then enters a paused state, so it cannot accept user input or display screens created by the developer. When the constructor finishes, the AMS calls method startApp (lines 78–82), which places the MIDlet in the active state, allowing the MIDlet to display content and accept user input. The MIDlet then waits for user input or another notification from the AMS. If the AMS calls method pauseApp (line 85), the MIDlet returns to the “paused state.” When the MIDlet is paused, the AMS must call method startApp to enable the MIDlet to reenter the active state. If the AMS calls method destroyApp (line 88) to clear the device’s memory for another application, the MIDlet’s execution terminates. Methods startApp, pauseApp and
**destroyApp** are abstract methods of class **MIDlet**, so every **MIDlet** subclass must override these methods.

Line 6 imports the J2ME CLDC I/O package that enable **TipTestMIDlet** to send and receive data from the servlets. Lines 9–11 import the MIDP packages for defining the MIDlet lifecycle, creating user interfaces and networking. J2ME divides the user-interface API between low-level and high-level APIs. The low-level API allows developers to incorporate graphics and shapes at precise pixel locations and to provide animation for applications such as games. The high-level user-interface API allows developers to incorporate text-fields, lists, forms and images for programs such as e-commerce applications and basic user interfaces.

```java
1 // TipTestMIDlet.java
2 // Receives TipTest from Servlet
3 package com.deitel.advjhtp1.wireless;

4 // J2ME Java package subset
5 import java.io.*;

6 // J2ME packages
7 import javax.microedition.midlet.*;
8 import javax.microedition.lcdui.*;
9 import javax.microedition.io.*;

10 public class TipTestMIDlet extends MIDlet {
11
12   private Display display; // display manager
13
14   // Screens displayed to user
15   private List mainScreen;
16   private List welcomeScreen;
17   private Form infoScreen;
18   private Form tipScreen;
19   private Form answerScreen;

20   // actions for soft-buttons
21   private Command selectCommand;
22   private Command nextCommand;
23   private Command backCommand;

24   private static final String servletBaseURL =
25       "http://localhost:8080/advjhtp1/";
26
27   private static final String welcomeServletName = "welcome";

28   // welcome servlet determines tip test servlet name
29   private String tipTestServletName;
30
31   private static final String welcomeServletURL =
32       servletBaseURL + welcomeServletName;
```

*Fig. 12.27 TipTestMIDlet downloads Tip Test from *TipTestServlet* (part 1 of 11).*
private String sessionID;

// constructor initializes display and main Screen
public TipTestMIDlet()
{
    // create soft button commands
    selectCommand = new Command( "Select", Command.OK, 0 );
    nextCommand = new Command( "Next Tip", Command.OK, 0 );
    backCommand = new Command( "Back", Command.BACK, 1 );

    // create main screen allowing welcome servlet connection
    mainScreen = new List( "TipTestMIDlet", List.IMPLICIT );
    mainScreen.addCommand( selectCommand );

    // allow soft button access for mainScreen
    mainScreen.setCommandListener(
        new CommandListener() {
            // invoked when user presses soft button
            public void commandAction( Command command, Displayable displayable )
            {
                // get data from welcome servlet
                String data = getServerData( welcomeServletURL );

                // create welcome Screen using servlet data
                display.setCurrent( createWelcomeScreen( data ) );
            }
        } // end anonymous inner class
    );

    // get appropriate Display for device
    display = Display.getDisplay( this );

} // end TipTestMIDlet constructor

// start MIDlet
public void startApp()
{
    // set display to main Screen
    display.setCurrent( mainScreen );
}

// pause MIDlet
public void pauseApp() {}

// destroy MIDlet
public void destroyApp( boolean unconditional ) {}
private Screen createWelcomeScreen( String data )
{
    String list[] = parseData( data, ';' );

    // create Screen welcoming user to tip test
    welcomeScreen = new List( list[ 0 ], List.IMPLICIT );

    welcomeScreen.append( "Take TipTest", null );
    welcomeScreen.addCommand( selectCommand );
    welcomeScreen.addCommand( backCommand );

    // get URL of information page
    final String url = new String( list[ 1 ].toCharArray() );

    // allow soft button access for welcomeScreen
    welcomeScreen.setCommandListener(
    new CommandListener() {

        // invoked when user presses soft button
        public void commandAction(
            Command command, Displayable displayable )
        {

            // soft button pressed is SELECT button
            if ( command.getCommandType() == Command.OK ) {

                // get data from static page
                String data =
                    getServerData( servletBaseURL + url );

                // display this data
                display.setCurrent( createInformationScreen( data ) );

            }

            // soft button pressed is BACK button
            else if ( command.getCommandType() == Command.BACK ) {
                display.setCurrent( mainScreen );
            }

        } // end method commandAction

    } // end anonymous inner class

    return welcomeScreen;
}

Fig. 12.27 TipTestMIDlet downloads Tip Test from TipTestServlet (part 3 of 11).
// create Screen showing servlets to which client can connect
private Screen createInformationScreen( String data )
{
    String list[] = parseData( data, ';' );

    // create Form showing available servlets
    infoScreen = new Form( "Information" );

    // create StringItem that provides directions
    StringItem infoTitle = new StringItem( list[ 0 ], null );
    infoScreen.append( infoTitle );

    // create ChoiceGroup allowing user to select servlet
    final ChoiceGroup choices = new ChoiceGroup( "",
        ChoiceGroup.EXCLUSIVE );
    choices.append( list[ 1 ], null );

    // append ChoiceGroup to Form
    infoScreen.append( choices );

    infoScreen.addCommand( selectCommand );
    infoScreen.addCommand( backCommand );

    // allow soft button access for this Screen
    infoScreen.setCommandListener(
        new CommandListener() {

            // invoked when user presses soft button
            public void commandAction( 
                Command command, Displayable displayable )
            {
                // soft button pressed is SELECT button
                if ( command.getCommandType() == Command.OK ) {

                    // user chooses which servlet to connect
                    int selectedIndex = choices.getSelectedIndex();
                    tipTestServletName =
                        choices.getString( selectedIndex );

                    // connect to servlet and receive data
                    String data = getServerData( servletBaseURL +
                        tipTestServletName );

                    // display next Screen according to data
                    display.setCurrent( createTipTestScreen( 
                        servletBaseURL + data ) );
                }
            }
        });
}
// soft button pressed is BACK button
else if ( command.getCommandType() == Command.BACK )
    display.setCurrent( welcomeScreen );

} // end method commandAction

} // end anonymous inner class

return infoScreen;

} // end method createInformationScreen

// create Screen to display Tip Test
private Screen createTipTestScreen( String data )
{
    // parse server data
    String list[] = parseData( data, '\n' );

    // create new Form to display test
tipScreen = new Form( "Tip Test" );

    // create image from server data
    Image serverImage = getServerImage( list[ 0 ] );

    // append image to Form
    if ( serverImage != null )
        tipScreen.append( serverImage );

    String choiceList[] = new String[ 4 ];

    // construct list for ChoiceGroup from data
    for ( int i = 0; i < choiceList.length; i++ )
        choiceList[ i ] = list[ i + 1 ];

    // create ChoiceGroup allowing user to input choice
    final ChoiceGroup choices = new ChoiceGroup( "Tip Test", ChoiceGroup.EXCLUSIVE, choiceList, null );

    // append ChoiceGroup to Form
    tipScreen.append( choices );

    tipScreen.addCommand( selectCommand );

    // allow soft button access for this Screen
tipScreen.setCommandListener(
        new CommandListener() {

            // invoked when user presses soft button
            public void commandAction(
                Command command, Displayable displayable )
            {
            }

        }
    );

Fig. 12.27 TipTestMIDlet downloads Tip Test from TipTestServlet (part 5 of 11).
Fig. 12.27 TipTestMIDlet downloads Tip Test from TipTestServlet (part 6 of 11).
// display next question
   display.setCurrent( createTipTestScreen(
       servletBaseURL + data );

   } // end anonymous inner class

   return answerScreen;

} // end method createAnswerScreen

// sends user's test selection to servlet
private String postData( int selection )
{
   // connect to server, then post data
   try {

      // connect to server sending User-Agent header
      HttpConnection httpConnection =
         ( HttpConnection ) Connector.open(
           servletBaseURL + tipTestServletName,
           Connector.READ_WRITE );

      setUserAgentHeader( httpConnection );

      // send sessionID, if one exists
      if ( sessionID != null )
         httpConnection.setRequestProperty(  
            "cookie", sessionID );

      // inform servlet of post request
      httpConnection.setRequestMethod( HttpConnection.POST );

      // open output stream to servlet
      DataOutputStream out =
         httpConnection.openDataOutputStream();

      // send selection
      out.writeUTF( Integer.toString( selection ) );
      out.flush();

      // get result from servlet
      String data = getData( httpConnection );

      httpConnection.close(); // close connection

      return data;

   } // end try

Fig. 12.27  TipTestMIDlet downloads Tip Test from TipTestServlet (part 7 of 11).
344     // handle if MIDlet cannot open HTTP connection
345     catch ( IOException ioException ) {
346         ioException.printStackTrace();
347     }
348
349     return null;
350 } // end method postData
351
352 // string tokenizer parses String into sub-string array
353 private String[] parseData( String data, char delimiter )
354 {
355     int newLines = 0;
356
357     // determine number of delimiter characters in String
358     for ( int i = 0; i < data.length(); i++ )
359         // increase number of delimiters by one
360         if ( data.charAt( i ) == delimiter )
361             newLines++;
362
363     // create new String array
364     String list[] = new String[ newLines ];
365
366     int oldNewLineIndex = 0;
367     int currentNewLineIndex;
368
369     // store Strings into array based on demiliter
370     for ( int i = 0; i < newLines; i++ ) {
371         // determine index where delimiter occurs
372         currentNewLineIndex =
373             data.indexOf( delimiter, oldNewLineIndex );
374
375         // extract String within delimiter characters
376         list[ i ] = data.substring( oldNewLineIndex,
377             currentNewLineIndex - 1 );
378
379         oldNewLineIndex = currentNewLineIndex + 1;
380     }
381
382     return list;
383 } // end method parseData
384
385 // connect to server and receive data
386 private String getServerData( String serverUrl )
387 {
388     // connect to server, then get data
389     try {
390         Fig. 12.27 TipTestMIDlet downloads Tip Test from TipTestServlet (part 8 of 11).
// connect to server sending User-Agent header
HttpConnection httpConnection = (HttpConnection) Connector.open(serverUrl);
setUserAgentHeader(httpConnection);

// send sessionID to server
if (sessionID != null)
    httpConnection.setRequestProperty("cookie", sessionID);

// get sessionID from server
String sessionIDHeaderField = httpConnection.getHeaderField("Set-cookie");

// store sessionID from cookie
if (sessionIDHeaderField != null) {
    int index = sessionIDHeaderField.indexOf(';');
    sessionID = sessionIDHeaderField.substring(0, index);
}

// receive server data
String data = getData(httpConnection);
httpConnection.close(); // close connection

return data;

} // end try

// handle exception communicating with HTTP server
catch (IOException ioException) {
    ioException.printStackTrace();
}

return null;

} // end method getServerData

// downloads an image from a server
private Image getServerImage(String imageUrl) {
    // download image
    try {

        // open connection to server
        HttpConnection httpConnection = (HttpConnection) Connector.open(imageUrl);

        int connectionSize = (int) httpConnection.getLength();
        byte imageBytes[] = new byte[connectionSize];

        Fig. 12.27 TipTestMIDlet downloads Tip Test from TipTestServlet (part 9 of 11).
// read image from server
InputStream input = httpConnection.openInputStream();
input.read( imageBytes );

// create image from imageBytes
return
    Image.createImage( imageBytes, 0, connectionSize );

// handle exception if InputStream cannot input bytes
catch ( IOException ioException ) {
    ioException.printStackTrace();
}
return null;

} // end method getServerImage

// set User-Agent header to identify client to servlet
private void setUserAgentHeader( HttpConnection httpConnection )
{
    // set User-Agent header
    try {
        // use profile/configuration properties for User-Agent
        String userAgentHeader = "Profile= " +
            System.getProperty( "microedition.profiles" ) +
            " Configuration=" +
            System.getProperty( "microedition.configuration" );

        // set header
        httpConnection.setRequestProperty( "User-Agent", userAgentHeader );
    }
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }
}

} // end method setUserAgentHeader

// open DataInputStream to receive data
private String GetData( HttpConnection httpConnection )
    throws IOException
{
    String data = ""; // stores data

Fig. 12.27 TipTestMIDlet downloads Tip Test from TipTestServlet (part 10 of 11).
Figure 12.28 shows a portion of the J2ME user-interface API. Each rectangle represents a class in the API. Each class with its name placed in italics is abstract, and the arrows represent inheritance relationships (an arrow points to the superclass). In the J2ME user-interface API, abstract superclass `Displayable` represents content that a MIDP-device can display on screen. The abstract superclasses `Screen` and `Canvas` both inherit from class `Displayable` and represent the high-level and low-level displayable content, respectively. Classes `Alert`, `Form`, `TextBox` and `List` are concrete subclasses of class `Screen`. An `Alert` is a `Screen` that the MIDlet displays for a brief period before displaying another `Screen`. A `Form` aggregates text-fields, images and group of selectable items for the user. A `TextBox` enables the user to input and edit text. A `List` is a list of `Strings` from which the user can select using the MIDP device’s keypad. Our case study uses classes `Form` and `List` to display information on screen. Class `Canvas` does not contain any subclasses. To use a `Canvas`, first create a concrete class that extends class `Canvas`, then override its `paint` method to draw graphics to the `Canvas`. Our case study incorporates only high-level user-interface classes, so we do not discuss how to use class `Canvas`.

**Portability Tip 12.1**

The J2ME low-level user-interface API helps developers create more visually appealing screens than does the high-level API. However, the low-level API does not guarantee layout congruity among several devices with different screen sizes. The high-level API provides a more consistent layout among devices.

Line 15 of class `TipTestMIDlet` declares an instance of class `Display`, which acts as a display manager for a MIDlet. A MIDlet must contain exactly one `Display` to display any `Displayable` object. Class `Display` is an example of the Singleton design pattern, which guarantees that a system instantiates a maximum of one object of a class. Occasionally, a system should contain exactly one object of a class—that is, once the system instantiates that object, the program should not be allowed to create additional

```java
496       // open input stream from connection
497       DataInputStream dataInputStream =
498           new DataInputStream(
499               httpConnection.openInputStream() );
500
501       int inputCharacter = dataInputStream.read();
502
503       // read all data
504       while ( inputCharacter != -1 ) {
505           data = data + ( char ) inputCharacter;
506           inputCharacter = dataInputStream.read();
507       }
508
509       dataInputStream.close(); // close stream
510
511       return data;
512
513   } // end method getData
514 }
```

Fig. 12.27 `TipTestMIDlet` downloads Tip Test from `TipTestServlet` (part 11 of 11).
objects of that class. Because a MIDP device has only one screen, a MIDlet should contain only one display manager to display the contents on that screen—therefore, only one \texttt{Display} object can exist in each MIDlet. The \texttt{static} method \texttt{getDisplay} of class \texttt{Display} returns a reference to the only \texttt{Display} object in the system—this \texttt{Display} object is also referred to as the singleton object. If the \texttt{Display} object has been created, subsequent calls to method \texttt{getDisplay} merely return the same reference to the singleton \texttt{Display} object.

The \texttt{Display} object ensures that only one of the \texttt{Screens} (\texttt{Displayable} subclasses) in \texttt{TipTestMIDlet} displays at a time. Lines 18–22 declare five \texttt{Screens} for \texttt{TipTestMIDlet}. \texttt{TipTestMIDlet} contains a \texttt{List} (line 18) to represent the main screen that contains a link to \texttt{WelcomeServlet}, a \texttt{List} (line 19) to show \texttt{index.txt}, a \texttt{Form} (line 20) to show \texttt{info.txt} (which provides the link to \texttt{TipTestServlet}), a \texttt{Form} (line 21) to show Tip-Test questions and a \texttt{Form} (line 22) to show Tip-Test answers.

\texttt{Screens} also have support for soft-buttons—buttons that are usually located below the display (but above the keypad) on wireless devices. In Fig. 12.8, the mouse cursor is hovering over the right soft button, which is highlighted. The word \texttt{Select} on the screen above the right soft button indicates that the user will select the highlighted \texttt{List} item when the user presses this soft button—the \texttt{Display} will then show a different \texttt{Screen}. J2ME provides this functionality to MIDlets through \texttt{Command} objects, which encapsulates an action to be executed by the object that receives the \texttt{Command} object. Lines 25–27 of \texttt{TipTestMIDlet} declare three \texttt{Command} objects—\texttt{selectCommand}, \texttt{nextCommand} and \texttt{backCommand}. The \texttt{selectCommand} object—as we will see momentarily—allows the user to select \texttt{List} items on the screen. The \texttt{nextCommand} object allows the user to receive the next tip from \texttt{TipTestServlet} when playing Tip Test. The \texttt{backCommand} object enables the user to view the previous \texttt{Screen}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{uml.png}
\caption{J2ME user-interface API class hierarchy.}
\end{figure}
Lines 46–48 instantiate these objects. The first argument to the `Command` constructor is the desired name—or label—to be displayed on the `Screen` above the soft button. The second argument is a constant that specifies how the MIDlet should respond after the user presses the soft button. For example, `Command.OK` indicates that the user has provided some input (via a text-field or via selecting a list item), `Command.BACK` indicates that the user should view the previous `Screen`. We program the logic that handles how the MIDlet behaves for each command type. The third argument indicates above which soft button the device will place the label. In a series of `Command` instantiations, the `Command` object with the lowest number has its label situated above the right soft button in the Sun MIDP-device emulator. For the object associated with the next lowest number, Sun’s device places its label above the left soft button. According to lines 46–48, the device places “Select” above the right soft button, “Next Tip” above the right soft button and “Back” above the left soft button.

**Portability Tip 12.2**

The `Command` numbering scheme varies from device-to-device. For example, the Sun emulator’s default location for `Command` text is above the right soft button. Other devices may situate the text above the left soft button.

Line 51 instantiates `List mainScreen`, which provides a link to `WelcomeServlet`. The first argument to the `List` represents the `List` name—this text appears at the top of the MIDP-device screen. The second argument is a constant that indicates the type of `List` instantiated. The `List` type determines how the user navigates the `List` with the keypad. This argument can assume one of three constant values—`List.IMPLICIT`, `List.EXCLUSIVE` and `List.MULTIPLE`. `List.IMPLICIT` indicates that the current focused item in the `List` is the user’s selection—i.e., the user changes the selection when scrolling among `List` items. `List.EXCLUSIVE` requires the user to press the center soft button to indicate a selection among `List` items, then press the right soft button to finalize the selection—however, before finalizing the selection (but after having marked the selection), the user can scroll among `List` items. `List.MULTIPLE` enables the user to select several items in the `List`.

Line 52 adds `selectCommand` to `mainScreen`, so the user will see “Select” above the right soft button. Lines 55–69 allow `mainScreen` to listen for events from `selectCommand` by creating a new `CommandListener` object. When the user presses a soft button, `selectCommand` invokes method `commandAction` (lines 59–67). This method takes as arguments the `Command` object associated with the recently pressed soft button and the `Displayable` object on which this action occurred. We explain method `commandAction`’s logic momentarily.

Line 73 gets the display manager for the device. When the constructor returns, the AMS invokes method `startApp`, which informs `TipTestMIDlet` to accept user input and to display screens. Using the display manager, line 81 sets the current display to the `mainScreen`. Figure 12.29 shows the output of this operation.

`TipTestMIDlet` now waits for user input. The only events registered to `TipTestMIDlet` originate from the right soft button—when the user presses this button, `selectCommand` calls method `commandAction` of any registered `CommandListener`. Line 63 passes `WelcomeServlet`’s URL to method `getServerData` (lines 390–432), which connects to a server and receives data. Lines 396–397 open an `HttpConnection` using the URL parameter. Line 398 calls method `setUserAgentHeader` (lines 466–488), so the
server can identify which client is making the request. Unlike the Internet Explorer, Open-
wave UP browser and Pixo clients, a J2ME client does not have a well-known User-Agent
header, so we must define our own header and store it in interface ClientUserAgent-
Headers, so our servlets can distinguish TipTestMIDlet as a J2ME client. Lines 473–
476 create a User-Agent header using TipTestMIDlet’s profile and configuration
information. Lines 479–480 assign this header to the HttpConnection request.

We also require a means to track this session so the servlet can maintain state informa-
tion—such as the correct answer—between sessions. Lines 406–407 store the HttpCon-
nection’s Set-cookie header field, which provides information on this session, in String sessionID (line 40). The Set-cookie header contains information delimited by semicolons. A server stores a session identifier before the first semicolon—lines 410–413 extract the session identifier. The next time TipTestMIDlet connects to the server, lines 401–403 send this information to identify the session.

Line 417 calls method getData (lines 491–513) to receive data from the server. Lines 497–499 open a DataInputStream to read the server-generated data. Lines 501–
507 read this data into a String, then return the String.

At this point, TipTestMIDlet has connected to WelcomeServlet and received data—which is index.txt—that represents the “welcome screen.” The data is

    eLearning Deitel Programming Tips ;j2me/info.txt ;

Line 66 passes this String as an argument to method createWelcomeScreen (lines 91–138). Line 93 calls method parseData (lines 354–387), which parses the data into an array of Strings, so we can access individual Strings. Method parseData acts
like the J2SE class `java.util.StringTokenizer`, which the J2ME `java.util` package does not contain (due to limited resources specified by MIDP requirements). We use a semicolon as the delimiter to parse the data, so method `parseData` returns a two-element `String` array that contains “eLearning Deitel Programming Tips” (the screen name) and “j2me/info.txt” (the link to Tip-Test directions). Line 96 creates `welcomeScreen` using the first element in the `String` array as the `List` name. Line 98 appends the `String` “Take TipTest” to the `List`, informing the user to download the Tip Test. Figure 12.30 shows `TipTestMIDlet`’s welcome screen.

Lines 99–134 register `welcomeScreen` as a `CommandListener` for events from `selectCommand` and `backCommand`. When the user presses a soft button, either `selectCommand` or `backCommand` calls method `commandAction` (lines 110–131), depending on which button the user pressed. The `Command` is an example of the Command Design Pattern, which we discussed in Section 2.3 when we presented Swing `Actions`. In J2ME, a `Command` object can contain various commands, or instructions, such as “show the next `Screen`,” “show the previous `Screen`,” or “exit the application.” Several `Displayable` objects in the system can listen for these commands—MIDP-application developers write the operations that each `Displayable` object performs when receiving `Commands`. For example, we register `welcomeScreen` as a listener for the `selectCommand` and `backCommand` object. When each `Screen` receives a `Command` event, method `commandAction` uses the `Command` event’s method `getCommandType` to determine if the command type is `Command.OK` (“show the next `Screen`”) or `Command.BACK` (“show the previous `Screen`”). We design `TipTestMIDlet` to act according to the command type received. If the user pressed the left soft button (`Back`), line 128 sets the `Display` to display `mainScreen`. If the user pressed the right soft button (`Select`), lines 117–122 draw the “information” `Screen`. Lines 117–118 call method `getServerData`, which connects to the server and receives `j2me/info.txt` as the `String`:

In this exercise, we will test your knowledge of the Deitel programming tips ;tiptest ;

Lines 121–122 pass this `String` as an argument to method `createInformationScreen` (lines 141–200), which constructs a screen that provides more information on Tip Test. Line 143 calls method `parseData` to parse the server-generated data into a two-element `String` array. `String` array’s first element represents the title of the `Screen` we will create, and the second element represents the link to `TipTestServlet`. Line 146 creates `infoScreen` as a `Form`. A `Form` uses `StringItems`—components that contain `Strings`—to display several lines of text. A `List` is able to display only one line of text and therefore would display only part of the `String` array’s first element. We use a `Form` to display the `String` array’s first element in its entirety. Lines 149–150 append a `StringItem` to `infoScreen` using the `String` array’s first element. Lines 153–158 create a `ChoiceGroup`—a group of items that the user can select in a `Form`—using the `String` array’s second element, so `TipTestMIDlet` can link to `TipTestServlet`. Note that line 154 declares the `ChoiceGroup` as `EXCLUSIVE`, so the user must specify which item to select—in our example, there exists only one choice (`tiptest`), so this item is selected by default. Figure 12.31 shows `TipTestMIDlet`’s information screen.
Fig. 12.30 TipTestMIDlet welcome screen. (Courtesy of Sun Microsystems, Inc.)

Fig. 12.31 TipTestMIDlet information screen. (Courtesy of Sun Microsystems, Inc.)
**Common Programming Error 12.2**

ChoiceGroups can be declared only as either EXCLUSIVE or MULTIPLE. Declaring a ChoiceGroup as INCLUSIVE results in an IllegalArgumentException.

Lines 160–196 register `infoScreen` as a CommandListener for events from `selectCommand` and `backCommand`. When the user presses a soft button, either `selectCommand` or `backCommand` calls method `commandAction` (lines 168–193), depending on which button the user pressed. If the user pressed the left soft button (Back), line 191 sets the Display to display `welcomeScreen`. If the user pressed the right soft button (Select), lines 175–186 display the “question” Screen. Line 175 determines which item in ChoiceGroup is selected—lines 177–178 assign this item to the TipTestServlet link. Lines 181–182 call method `getServerData`, which connects to TipTestServlet and receives Tip Test. TipTestServlet generates random information each time TipTestMIDlet establishes a connection; the data that TipTestServlet generates appears in the following format:

```
http://localhost:8080/advjhtp1/j2me/png/portability.png
PERF
CPE
TAD
PORT
```

Lines 185–186 call method `createTipTestScreen` (lines 203–256) to display the Tip-Test question. Line 206 calls method `parseData` to parse the Tip Test into a five-element `String` array. The first element contains the image file name located on the server. The remaining four elements contain the four tip abbreviations from which the user must choose the correct answer. We create a Form, so we can show an Image and a ChoiceGroup—no other Displayable subclass offers this functionality. Line 209 creates a new Form—`tipScreen`—to show Tip Test.

Line 212 passes the String array’s first element to method `getServerImage` (lines 435–463), which creates an Image object from the image file on the server. Lines 441–442 create an HttpConnection to the server, lines 448–449 read the data into an InputStream and lines 452–453 return an Image from the InputStream data.

Lines 215–216 append the Image to `tipScreen`. We now represent the four possible answers associated with this Image as a ChoiceGroup. Lines 218–222 create a String array to hold each of the four String answers. Lines 225–226 instantiate the ChoiceGroup, and line 229 appends this ChoiceGroup to `tipScreen`. Note that the ChoiceGroup’s constructor takes four arguments (unlike the ChoiceGroup constructor in method `createInformationScreen`, which took two arguments). The first argument represents the ChoiceGroups name, which we title “Tip Test.” The second argument declares this ChoiceGroup as EXCLUSIVE, so the user must indicate the selection before proceeding. The third argument is the String array that contains each possible String answer. The fourth argument represents an array of Images associated with the String array—the device places each Image in the Image array next to the respective String in the String array. We pass a null value for this argument, because we do not need to display any Images next to the four possible answers.

Figure 12.32 shows `tipScreen`. The picture on the left shows the Image for which the user must guess the correct name. When the user scrolls down on this screen (picture
on the right), the user will see the four possible answers. The user must press the center soft button to indicate the selection.

Lines 231–251 register `tipScreen` as a `CommandListener` for events from `selectCommand`. When the user presses the `Select` button, `selectCommand` calls method `commandAction` (lines 238–249). Line 242 determines which item in the `ChoiceGroup` has been selected—this item represents the user’s selection. Line 244 passes the selection to method `postData` (lines 306–351), which sends the selection to `TipTestServlet`. Lines 312–315 connect to `TipTestServlet` and specify that the connection is bidirectional, so `TipTestMIDlet` can send data to, and receive data from, `TipTestServlet`. Lines 312–315 connect to `TipTestServlet` and specify that the connection is bidirectional, so `TipTestMIDlet` can send data to, and receive data from, `TipTestServlet`. Line 317 sets the `User-Agent` header so `TipTestServlet` can distinguish `TipTestMIDlet` as a J2ME client. Lines 320–322 send the session identifier that method `getServerData` stored to identify the session. Line 325 specifies that `TipTestServlet` will receive `postData` data from `TipTestMIDlet`. Lines 328–329 open a `DataOutputStream`, through which line 332 sends the user selection to `TipTestServlet`. As we discussed in Section 12.3.4, `TipTestServlet` sends the correct answer to the J2ME client upon receiving this data. Line 336 calls method `getData` to receive this `String` data from the servlet. This data will have the following format:

```
Correct
Portability Tip
Organizations that develop software must often produce versions customized to a variety of computers and operating systems. These tips offer suggestions to make your applications more portable.
```
Line 338 closes the connection between TipTestMIDlet and TipTestServlet, and line 340 returns the data that contains the correct answer. Lines 247–248 pass this String data to method createAnswerScreen (lines 259–303), which creates the Screen that displays the answer. Line 262 calls method parseData to parse the data into a three-element String array. The first element represents whether the user was correct or incorrect. The second and third elements hold the correct tip name and description, respectively. Line 265 instantiates Form answerScreen by passing the String array’s first element to the constructor—we use a Form to show the tip description in its entirety. Lines 268–273 instantiate two String items to hold the tip name and description. Lines 276–277 append these String items to answerScreen. Figure 12.33 shows answerScreen. The picture on the left shows the screen that contains the correct tip name and description. When the user scrolls down on this screen, the user will see the right picture, which shows the remainder of the tip description.

Lines 279–299 register tipScreen as a CommandListener for events from nextCommand. The nextCommand object’s behavior is identical to that of the selectCommand object; however, because we cannot alter selectCommand’s label to hold “Next Tip” as a value, we had to instantiate another Command object to ensure that the “Next Tip” text appears above the right soft button. When the user presses the right soft button, nextCommand calls method commandAction (lines 286–296), which calls method createTipTestScreen to generate another Tip-Test question—users can play Tip Test for as long as they desire.

---

Fig. 12.33 TipTestMIDlet Tip-Test answer screen. (Courtesy of Sun Microsystems, Inc.)
This concludes our case study of Java-based wireless application development and Java 2 Micro Edition. In this section, we created a three-tier architecture in which TipTestServlet (middle tier) marked up a randomly generated Tip-Test question as XML, applied an XSLT to the XML document, then sent the resulting document to clients. We then introduced J2ME by discussing the CLDC and MIDP—the fundamental J2ME APIs for building applications to run on mobile devices. We studied a MIDlet’s lifecycle and examined how to create a new MIDlet from class MIDlet. We created a MIDP application—TipTestMIDlet—then discussed how it retrieves data from WelcomeServlet and TipTestServlet. We also discussed how TipTestMIDlet uses this data to construct user interfaces and how, by using Command objects, a TipTestMIDlet enables the user to navigate through various Screens.

In Chapter 13, we continue our discussion by presenting Remote Method Invocation (RMI), which enables client objects to call methods of objects on other systems. In Chapter 14, we begin discussing Enterprise JavaBeans (EJBs), which provide a model for building business logic in enterprise Java applications.

12.5 Installation Instructions

This section provides installation instructions for the software in this case study.

Web Server Configuration

This case study requires a web server that can run servlets. We recommend the Apache Tomcat Server. We provided directions to install Tomcat in Section 9.3.1. When you have installed Tomcat, copy the contents of the advjhtp1 directory on the CD to the advjhtp1 directory in the Tomcat directory on your system. The advjhtp1 directory on the CD contains four directories—iMode, j2me, XHTML, WAP—that contain content that the servlets distribute to each client.

Next, you must copy the contents of Fig. 12.34, which we provide on the CD, in web.xml’s web-app element. The web.xml file should exist in the WEB-INF Tomcat directory. For example, on our system, web.xml is located in

\`C:\jakarta-tomcat-3.2.2\webapps\advjhtp1\WEB-INF\`

Fig. 12.34 Deployment descriptor to run WelcomeServlet and TipTestServlet (part 1 of 2).
In Fig. 12.34, lines 2–13 describe WelcomeServlet, and line 15–38 describe TipTestServlet. Lines 23–33 declare two init-param elements that enable us to change the database that TipTestServlet uses, without having to modify TipTestServlet.java. Lines 41–49 map the two servlets to specified URLs. Lines 51–59 con-
figure Tomcat to serve WML content with correct MIME types—if we do not specify these MIME types, the WAP clients will not receive content.

The last step for configuring Tomcat to run the case study involves specifying those classes that Tomcat uses to run the servlets. Create the directory structure `com/deitel/advjhtpl/wireless` in the WEB-INF/classes directory. For example, on our system, the directory structure is

```
C:\jakarta-tomcat-3.2.2\webapps\advjhtpl\WEB-INF\classes\com\deitel\advjhtpl\wireless
```

Copy `WelcomeServlet.class` and `TipTestServlet.class` into this directory.

**Database Configuration**

This case study also requires a database from which the servlet can extract tip information. We recommend the Cloudscape database, which we provide installation instructions in Sections 8.1 and 8.5. Copy the `tips.sql` file and the `tips` directory from our CD to the `frameworks/RmiJdbc/bin` directory where you have installed Cloudscape on your system. For example, `C:\cloudscape_3.6` is the directory that contains Cloudscape on our system, so

```
C:\cloudscape_3.6\frameworks\RmiJdbc\bin
```

should contain `tips.sql` and the `tips` directory. Next, copy `cloudscape.jar` and `RmiJdbc.jar` (located on the CD) into the `WEB-INF/lib` Tomcat directory. For example, on our system, this directory is

```
C:\jakarta-tomcat-3.2.2\webapps\advjhtpl\WEB-INF\lib
```

Placing `cloudscape.jar` and `RmiJdbc.jar` in this directory enable `TipTestServlet` to connect to, and extract from, a Cloudscape database.

**J2ME Wireless Toolkit Installation and Configuration**

To use the Sun MIDP-device emulator and develop MIDP applications, you must use the J2ME Wireless Toolkit. We include this toolkit on the CD, or you may download the toolkit from

```
java.sun.com/products/j2mewtoolkit/
```

At the time of this writing, Release 1.0.2 is available for download. In the installation procedure, you can specify whether you want to integrate the J2ME Wireless Toolkit with Forte—which facilitates developing MIDP applications in Forte—or to run the toolkit as a “stand-alone” application. We recommend that you integrate the toolkit in Forte.

If you integrate the toolkit in Forte, create the directory structure `com/deitel/advjhtpl/wireless` in your system’s Forte development directory. For example, `C:\forte4j\Development` is the Forte development directory on our system, so we create the directory structure

```
C:\forte4j\Development\com\deitel\advjhtpl\wireless
```
Next, copy `TipTestMIDlet.java` and `TipTestMIDlet.jar` from our CD to this directory. When you expand this directory structure in Forte, you should see

```
TipTestMIDlet.java
TipTestMIDlet.jar
```

`TipTestMIDlet.jar` is the MIDlet suite that contains class `TipTestMIDlet`. To run class `TipTestMIDlet`, right-click on the `TipTestMIDlet.jar` icon, then select execute.

If you prefer to use the Wireless Toolkit (or if you had installed the toolkit as a stand-alone application), first open the Wireless Toolkit by selecting `KToolbar` in the directory where you installed the toolkit. Next, press the `New Project` button. In the `Project Name` text field, type

```
TipTestMIDlet
```

In the `MIDlet Class Name` text field, type

```
com.deitel.advjhtp1.wireless.TipTestMIDlet
```

then press the `Create Project` button. When the `Settings` frame appears, press the `OK` button. Next, copy `TipTestMIDlet.java` from our CD to the `apps/TipTestMIDlet/src` directory in the directory where you installed the toolkit. For example, `C:\J2mewtk` is the directory where we installed the toolkit on our system, so we copy `TipTestMIDlet.java` to

```
C:\J2mewtk\apps\TipTestMIDlet\src\TipTestMIDlet.java
```

Return to the Wireless toolkit, and press the `Build` button. When `TipTestMIDlet` has finished compiling and preverifying, press the `Run` button to execute `TipTestMIDlet`. The `Device` menu—located on the toolkit’s right side—enables you to select from among several devices to run `TipTestMIDlet`. Although the default device is the Sun grayscale MIDP-device emulator, you can specify to run `TipTestMIDlet` on other devices, such as RIM’s Blackberry-957™ and the Motorola i85s™.

### Other clients

Figure 12.35 lists the URLs to download the browsers used in this case study:

<table>
<thead>
<tr>
<th>Browser</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Openwave UP simulator</td>
<td><a href="developer.openwave.com/download/license_41.html">developer.openwave.com/download/license_41.html</a></td>
</tr>
<tr>
<td>Pixo Internet Microbrowser</td>
<td><a href="www.pixo.com/products/products001.htm">www.pixo.com/products/products001.htm</a></td>
</tr>
</tbody>
</table>

Fig. 12.35 Case-study browser URLs.
12.6 Internet and World Wide Web Resources

www.java.sun.com/j2me
This site contains Sun’s Java 2 Micro Edition available for download.

www.onjava.com/pub/a/onjava/2001/03/08/J2ME.html
This site contains a J2ME article.

www.jguru.com/faq/home.jsp?topic=J2ME
This site presents the J2ME FAQ.

www.wirelessdevnet.com/channels/java/features/j2me_http.phtml
This site discusses network programming with J2ME devices.

www.ericrogiguere.com/microjava/fallacies.html
This site lists some common misconceptions of J2ME.

www.motorola.com/java
This site discusses J2ME integration in new Motorola’s wireless devices.

www.internetnews.com/wd-news/article/0,,10_533091,00.html
This site discusses J2ME integration in the Palm devices.

www.mot.com/java/devices.html
This site lists several wireless devices that use J2ME technology in the market.

www.nttdocomo.com/i/index.html
This is the Web site of NTT DoCoMo—the creators of i-mode.

www.anywhereyougo.com/imode/Index.po
This site presents news about i-mode development.

www.i-modesales.com
This site offers information on i-mode phones and services.

www.wap.com
This site offers news about WAP development.

www.wapforum.org/
This site discusses advancements in the WAP field.

SUMMARY

- This case study is a three-tier architecture of a multiple choice test (Tip Test) that allows users to test their knowledge of Deitel programming tips.
- Developers can use Java technology to develop wireless and server-side applications that developers can also develop with ASP.
- The information tier consists of a database. The middle tier consists of two servlets—WelcomeServlet and TipTestServlet—that generate content to clients of various types. The client tier consists of four client types—Internet Explorer, WAP, i-mode and J2ME. Each client type renders content differently.
- Microsoft Internet Explorer receives XHTML content.
- The Openwave UP simulator is the WAP client that receives WML content.
- The Pixo Internet Microbrowser is the i-mode client that receives cHTML content.
- The Sun MIDP-device emulator displays a J2ME client that receives content in plain-text format.
- J2ME™ (Java™ 2 Micro Edition) is Sun’s newest Java platform for developing applications for various consumer devices, such as set-top boxes, Web terminals, embedded systems, mobile phones and cell pagers.
MIDP (Mobile Information Device Profile) is a set of APIs that allow developers to handle mobile-device-specific issues, such as creating user interfaces, storing information locally and networking.

Devices that run applications for MIDP are called MIDP devices (e.g., cell phones or pagers).

Clients interact with servlets by making a series of `get` and `post` request to the servlets. When a client sends a `get` request to an `HttpServlet`, method `doGet` handles the request. When a client sends a `post` request to an `HttpServlet`, method `doPost` handles the request.

All clients contain a unique `User-Agent` header, which contains information on what type of client is requesting data from the server.

Method `getInitParameter` of `HttpServlet` allows us to specify information (such as database locations or drivers) declared in `web.xml`. We need only to modify the `<param-value>` element in the `<init-param>` element in `web.xml` to change this information.

`org.w3c.dom.Element` objects element nodes in XML documents.

We can apply an XSLT to an XML `Document` to produce different content for each different client type. Each client type renders this content accordingly.

We did not use XML for our J2ME client, because at the time of this writing, J2ME cannot support XML without using non-standard XML-based software.

J2ME uses the Connected Limited Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP), which offer developers a set of APIs to write J2ME applications and deploy them across several mobile devices.

The CLDC is a set of APIs that allow developers to create applications for devices that have limited resources—i.e., limited screen size, memory, power and bandwidth.

Currently, the CLDC does not contain certain features that developers often “take for granted” in other Java platforms—e.g., floating-point operations, serializable objects and thread groups.

The MIDP is a set of APIs that allow developers to handle mobile-device-specific issues, such as creating user interfaces, permitting local storage and defining the lifecycles of MIDP-client applications.

A MIDlet is a type of MIDP-client application. All MIDlets must extend class `javax.microedition.midlet.MIDlet`.

MIDP developers store several MIDlets in a jar file—called a MIDlet suite—on a server.

The application management software (AMS) on the MIDP device downloads the MIDlet suite from the server, opens the MIDlet suite, then launches the user-specified MIDlet on the MIDP device.

The AMS uses an application descriptor file to load the MIDlet application. This file contains various information on the MIDlet suite and the MIDP device.

The MIDlet lifecycle consists of methods `startApp`, `pauseApp` and `destroyApp`.

J2ME divides the user-interface API between low-level and high-level APIs. The low-level API allows developers to incorporate graphics and provide animation, whereas the high-level user-interface API allows developers to incorporate text fields, lists and forms.

Class `Displayable` represents content that a MIDP-device can display on screen. Classes `Screen` (high-level) and `Canvas` (low-level) extend class `Displayable`.

Class `Display` acts as a display manager for a MIDlet. Exactly one `Display` object can exist in a MIDlet.

J2ME provides “soft-button support” in MIDlets through `Command` objects, which encapsulates an action to be executed by the object that receives the `Command` object.

`Lists` and `Forms` are two high-level user-interface classes that extend class `Screen`.

A `Form` can include `ChoiceGroups`—groups of items the user can select in a `Form`. 
**TERMINOLOGY**

application management software (AMS)  J2ME low-level user-interface API
chHTML (compact HTML)  Java application descriptor file
class Alert  KVM
class Canvas  Openwave UP simulator
class ChoiceGroup  method `commandAction` of class
  `Command`  `CommandListener`
class Command  method `doGet` of class `HttpServlet`
class CommandListener  method `doPost` of class `HttpServlet`
class Display  method `destroyApp` of class MIDlet
class Displayable  method `pauseApp` of class MIDlet
class DocumentBuilder  method `startApp` of class MIDlet
class DocumentBuilderFactory  middle tier
class DOMSource  MIDlet
  `CommandListener`
class Element  MIDlet lifecycle
class Form  MIDP application
class HttpServlet  MIDP devices
class HttpServletRequest  MIDP package `javax.microedition.io`
class HttpServletResponse  MIDP package
  `javax.microedition.lcdui`
class InputStream  MIDP package `javax.microedition.midlet`
class List  Mobile Information Device Profile (MIDP)
class Screen  namespace
class StreamSource  Pixo Internet Microbrowser
class StringItem  server
  `javax.microedition.rms`
class TextBox  Sun MIDP-device emulator
class Transformer  three-tier architecture
class TransformerFactory  `User-Agent` header
class Client  `javax.microedition.midlet.rms`
CLDC packages `java.io`, `java.lang`  mobile device
class `javax.microedition.lcdui`  Mobile Information Device Profile (MIDP)
class `java.util`  namespace
class `javax.microedition.midlet`  Pixo Internet Microbrowser
class `javax.microedition.rms`  server
get and `post` data  Sun MIDP-device emulator
i-mode  three-tier architecture
database  `User-Agent` header
Extensible HyperText Markup (XHTML)  `javax.microedition.midlet.rms`
Language (XHTML)  mobile device
`get` and `post` data  Mobile Information Device Profile (MIDP)
i-mode  namespace
information tier  `javax.microedition.rms`
Java 2 Micro Edition (J2ME)  `javax.microedition.midlet.rms`
J2ME high-level user-interface API  `javax.microedition.midlet`

**SELF-REVIEW EXERCISES**

12.1 Fill in the blanks in each of the following statements:

a) Typically, a three-tier architecture contains a ________ layer, a ________ layer and a ________ layer.

b) The servlets in our case study generated ________ content to Internet Explorer, ________ content to the WAP client, ________ content to the i-mode client and ________ content to the J2ME client.

c) `TipTestServlet` used a ________ object to store the correct tip name and description.
d) CLDC stands for ________.
e) MIDP stands for ________.
f) Developers can package several MIDlets in a jar file called a ________.
g) The application descriptor file contains the ________ extension.
h) The MIDlet lifecycle consists of methods ________, ________ and ________.
i) In package javax.microedition.lcdui, class ________ represents content that a MIDP device can display on screen.
j) Class javax.microedition.Display is an example of the ________ design pattern.

12.2 State whether each of the following statements is true or false. If false, explain why.

a) An HttpServlet’s calls method doGet upon receiving a get request and method doPost upon receiving a post request. (true/false).

b) A User-Agent header contains information about the server. (true/false).


d) In this case study, TipTestServlet uses XSLT to transform XML Tip Test Documents to well formed content for all clients. (true/false).

e) The J2ME CLDC package consist of java.io, java.lang and java.net. (true/false).

f) The application management software loads a MIDlet on a MIDP device. (true/false).

g) The application description file specifies such information as the MIDP device’s configuration and the MIDlet’s name. (true/false).

h) Classes Alert, Form, Screen, and List are the concrete classes of the MIDP high-level user-interface package. (true/false).

i) A javax.microedition.lcdui.Command object encapsulates an action to be executed by the object that received the Command. (true/false).

j) An HttpSession’s Set-Cookie header field contains session information. (true/false).

12.1 a) client (top), middle, information (bottom). b) XHTML, WAP, cHTML, plain text. c) HttpSession. d) Connected Limited Device Configuration. e) Mobile Information Device Profile. f) MIDlet suite. g) .jad. h) startApp, pauseApp, destroyApp. i) Displayable. j) Singleton.

12.2 a) True. b) False. A User-Agent header contains information on what client type is requesting data from a server. c) False. DocumentBuilderFactory’s static method newDocumentBuilder creates a new DocumentBuilder. Class DocumentBuilder creates new Documents. d) False. TipTestServlet uses XSLT to transform XML Tip Test Documents to well-formed content for only Internet-Explorer, WAP and i-mode clients. e) False. The J2ME CLDC package consist of java.io, java.lang and java.util. f) True. g) True. h) Classes Alert, Form, TextBox, and List are the concrete classes of the MIDP high-level user-interface package. Class Screen is an abstract class these classes extend. i) True. j) True.

EXERCISES

12.3 To make TipTestServlet more manageable and extendable, we stored the database driver name and URL in the <init-param> element in web.xml. List 3 other constants that we could have stored from WelcomeServlet and TipTestServlet in the <init-param> element.

12.4 Extend the list of possible answers in the Tip-Test question from four to five.
12.5 Method `getResultTable` of `TipTestServlet` creates a two-dimensional `String` array `ResultSet` representation. However, because these dimensions are pre-defined (seven rows by five columns), the `String` array cannot store additional rows if we append another tip to the database. Modify `TipTestServlet` so that it can handle any number of tips from the database. Use techniques shown in Chapter 8, Java Database Connectivity (JDBC), to create a scrollable `ResultSet` so that you can determine the number of rows in the `ResultSet` and size the `String` array accordingly.

12.6 Lines 29–38 of class `TipTestMIDlet` specify the servlet URL and the `WelcomeServlet` URL. This approach can be problematic if a network administrator changes either URL. In practice, we would like the application descriptor file to contain the servlet URL and the MIDP-device user to specify the `WelcomeServlet` URL with the MIDP-device keypad.

a) Encode the servlet URL—`http://localhost:8080/advjhtpl/`—in the `TipTestMIDlet` application descriptor file by typing

```
Servlet-URL:http://localhost:8080/advjhtpl/
```

at the end of `TipTestMIDlet`’s `.jad` file. In `TipTestMIDlet`, use method `getAppProperty` of class `MIDlet` to return the servlet URL. (Method `getAppProperty` takes a property-tag `String`—e.g., "Servlet-URL"—as an argument and returns the `String` associated with that property.)

b) Enable the user to specify the `WelcomeServlet` URL. Modify the class `TipTestMIDlet` constructor to create `mainScreen` (line 51) as a `TextBox`—a `Screen` subclass that enables the user to input text. A `TextBox` constructor takes four arguments: 1) a `String` that represents the `TextBox`’s title; 2) a `String` that represents the `TextBox`’s initial contents; 3) an `int` that represents the maximum number of characters allowed in the `TextBox`; 4) an input constraint `int` that specifies the required input format—e.g., if you want the user to enter a phone number, the constraint is `TextField.PHONENUMBER`. We want the user to enter a URL, so the constraint is `TextField.URL`. Use method `getString` of class `TextBox` to return the `TextBox`’s contents (user input) to store the `WelcomeServlet` URL when the user presses the Select soft button.

12.7 Modify `TipTestMIDlet` to enable the user to exit the application in the Tip-Test answer screen in addition to receiving the next Tip-Test question. Create a new `Command` object that uses `Command.EXIT` and register `answerScreen` as a listener for this `Command`. (Hint: override method `destroyApp` to call method `notifyDestroy` of class `javax.microedition.midlet.MIDlet`.

**BIBLIOGRAPHY**


Remote Method Invocation

Objectives

- To understand the distributed computing concepts.
- To understand the architecture of RMI.
- To be able to use activatable RMI objects to build resilient distributed systems.
- To understand how to use RMI callbacks.
- To be able to build RMI clients that download necessary classes dynamically.
- To be able to build activatable RMI objects.

*Dealing with more than one client at a time is the business world’s equivalent of bigamy. It’s so awkward to tell one client that you’re working on someone else’s business that you inevitably start lying.*

Andrew Frothingham

*They also serve who only stand and wait.*

John Milton

*Rule 1: The client is always right.*

*Rule 2: If you think the client is wrong, see Rule 1.*

Sign seen in shops

*I love being a writer. What I can’t stand is the paperwork.*

Peter De Vries
Chapter 13 Remote Method Invocation 791

13.1 Introduction

In this chapter, we introduce Java’s distributed computing capabilities with Remote Method Invocation (RMI). RMI allows Java objects running on separate computers or in separate processes to communicate with one another via remote method calls. Such method calls appear to the programmer the same as those operating on objects in the same program.

RMI is based on a similar, earlier technology for procedural programming called remote procedure calls (RPCs) developed in the 1980s. RPC allows a procedural program (i.e., a program written in C or another procedural programming language) to call a function residing on another computer as conveniently as if that function were part of the same program running on the same computer. A goal of RPC was to allow programmers to concentrate on the required tasks of an application by calling functions, while making the mechanism that allows the application’s parts to communicate over a network transparent to the programmer. RPC performs all the networking and marshaling of data (i.e., packaging of function arguments and return values for transmission over a network). A disadvantage of RPC is that it supports a limited set of simple data types. Therefore, RPC is not suitable for passing and returning Java objects. Another disadvantage of RPC is that it requires the programmer to learn a special interface definition language (IDL) to describe the functions that can be invoked remotely.

RMI is Java’s implementation of RPC for Java-object-to-Java-object distributed communication. Once a Java object registers as being remotely accessible (i.e., it is a remote object), a client can obtain a remote reference to that object, which allows the client to use that object remotely. The method call syntax is identical to the syntax for calling methods of other objects in the same program. As with RPC, RMI handles the marshaling of data across the network. However, RMI also enables Java programs to transfer complete Java objects using Java’s object-serialization mechanism. The programmer need not be concerned with the transmission of the data over the network. RMI does not require the programmer to learn an IDL, because the J2SE SDK includes tools for generating all the

Outline

| 13.1 | Introduction |
| 13.2 | Case Study: Creating a Distributed System with RMI |
| 13.3 | Defining the Remote Interface |
| 13.4 | Implementing the Remote Interface |
| 13.5 | Compiling and Executing the Server and the Client |
| 13.6 | Case Study: Deitel Messenger with Activatable Server |
| 13.6.1 | Activatable Deitel Messenger ChatServer |
| 13.6.2 | Deitel Messenger Client Architecture and Implementation |
| 13.6.3 | Running the Deitel Messenger Server and Client Applications |
| 13.7 | Internet and World Wide Web Resources |

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises
networking code from the program’s interface definitions. Also, because RMI supports only Java, no language-neutral IDL is required; Java’s own interfaces are sufficient.

We present two substantial RMI examples and discuss the key concepts of RMI as we encounter them throughout the examples. After studying these examples, you should have an understanding of the RMI networking model and should be able to take advantage of advanced RMI features for building Java-to-Java distributed applications.

[Note: For Java-to-non-Java communication, you can use Java IDL (introduced in Java 1.2) or RMI-IIOP. Java IDL and RMI-IIOP enable applications and applets written in Java to communicate with objects written in any language that supports CORBA (Common Object Request Broker Architecture). Please see Chapter 26, CORBA: Part 1 and Chapter 27, CORBA: Part 2 for our discussion of CORBA and RMI-IIOP.]

13.2 Case Study: Creating a Distributed System with RMI

In the next several sections, we present an RMI example that downloads the Traveler’s Forecast weather information from the National Weather Service Web site:

http://iwin.nws.noaa.gov/iwin/us/traveler.html

[Note: As we developed this example, the format of the Traveler’s Forecast Web page changed several times (a common occurrence with today’s dynamic Web pages). The information we use in this example depends directly on the format of the Traveler’s Forecast Web page. If you have trouble running this example, please refer to the FAQ page on our Web site, www.deitel.com.]

We store the Traveler’s Forecast information in an RMI remote object that accepts requests for weather information through remote method calls.

The four major steps in this example include:

1. Defining a remote interface that declares methods that clients can invoke on the remote object.
2. Defining the remote object implementation for the remote interface. [Note: By convention, the remote object implementation class has the same name as the remote interface and ends with Impl.]
3. Defining the client application that uses a remote reference to interact with the interface implementation (i.e., an object of the class that implements the remote interface).
4. Compiling and executing the remote object and the client.

13.3 Defining the Remote Interface

The first step in creating a distributed application with RMI is to define the remote interface that describes the remote methods through which the client interacts with the remote object using RMI. To create a remote interface, define an interface that extends interface java.rmi.Remote. Interface Remote is a tagging interface—it does not declare any methods, and therefore places no burden on the implementing class. An object of a class that implements interface Remote directly or indirectly is a remote object and can be accessed—with appropriate security permissions—from any Java virtual machine that has a connection to the computer on which the remote object executes.
Software Engineering Observation 13.1

Every remote method must be declared in an interface that extends \texttt{java.rmi.Remote}.

Software Engineering Observation 13.2

An RMI distributed application must export an object of a class that implements the \texttt{Remote} interface to make that remote object available to receive remote method calls.

Interface \texttt{WeatherService} (Fig. 13.1)—which extends interface \texttt{Remote} (line 10)—is the remote interface for our remote object. Line 13 declares method \texttt{getWeatherInformation}, which clients can invoke to retrieve weather information from the remote object. Note that although the \texttt{WeatherService} remote interface defines only one method, remote interfaces can declare multiple methods. A remote object must implement all methods declared in its remote interface.

When computers communicate over networks, there exists the potential for communication problems. For example, a server computer could malfunction, or a network resource could malfunction. If a communication problem occurs during a remote method call, the remote method throws a \texttt{RemoteException}, which is a checked exception.

Software Engineering Observation 13.3

Each method in a \texttt{Remote} interfaces must have a \texttt{throws} clause that indicates that the method can throw \texttt{RemoteException}.

Software Engineering Observation 13.4

RMI uses Java’s default serialization mechanism to transfer method arguments and return values across the network. Therefore, all method arguments and return values must be \texttt{Serializable} or primitive types.

13.4 Implementing the Remote Interface

The next step is to define the remote object implementation. Class \texttt{WeatherServiceImpl} (Fig. 13.2) is the remote object class that implements the \texttt{WeatherService} remote interface. The client interacts with an object of class \texttt{WeatherServiceImpl} by invoking

```java
// WeatherService.java
// WeatherService interface declares a method for obtaining weather information.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.rmi.*;
import java.util.*;

public interface WeatherService extends Remote {
    // obtain List of WeatherBean objects from server
    public List getWeatherInformation() throws RemoteException;
}
```

Fig. 13.1 \texttt{WeatherService} interface.
method `getWeatherInformation` of interface `WeatherService` to obtain weather information. Class `WeatherServiceImpl` stores weather data in a `List` of `WeatherBean` (Fig. 13.3) objects. When a client invokes remote method `getWeatherInformation`, the `WeatherServiceImpl` returns a reference to the `List` of `WeatherBeans`. The RMI system returns a serialized copy of the `List` to the client. The RMI system then de-serializes the `List` on the receiving end and provides the caller with a reference to the `List`.

The National Weather Service updates the Web page from which we retrieve information twice a day. However, class `WeatherServiceImpl` downloads this information only once, when the server starts. The exercises ask you to modify the server to update the data twice a day. [Note: `WeatherServiceImpl` is the class affected if the National Weather Service changes the format of the `Traveler’s Forecast` Web page. If you encounter problems with this example, visit the FAQ page at our Web site `www.deitel.com`]

```java
// WeatherServiceImpl.java
// WeatherServiceImpl implements the WeatherService remote
// interface to provide a WeatherService remote object.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.io.*;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class WeatherServiceImpl extends UnicastRemoteObject
        implements WeatherService {
    private List weatherInformation;  // WeatherBean object List

    // initialize server
    public WeatherServiceImpl() throws RemoteException
    {
        super();
        updateWeatherConditions();
    }

    // get weather information from NWS
    private void updateWeatherConditions()
    {
        try {
            System.out.println("Update weather information...");

            // National Weather Service Traveler’s Forecast page
            URL url = new URL("http://iwin.nws.noaa.gov/iwin/us/traveler.html");
```

Fig. 13.2 `WeatherServiceImpl` class implements remote interface `WeatherService` (part 1 of 3).
BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));

String separator = "TAV12";

String dayHeader = "CITY WEA HI/LO WEA HI/LO";
String nightHeader = "CITY WEA LO/HI WEA LO/HI";

String inputLine = " ";

weatherInformation = new ArrayList();

// create WeatherBean object for city. First 16
// characters are city name. Next, six characters
// are weather description. Next six characters
// are HI/LO or LO/HI temperature.
WeatherBean weather = new WeatherBean(
    inputLine.substring(0, 16),
    inputLine.substring(16, 22),
    inputLine.substring(23, 29));

// add WeatherBean to List
weatherInformation.add(weather);

Fig. 13.2 WeatherServiceImpl class implements remote interface WeatherService (part 2 of 3).
Remote Method Invocation

Class **WeatherServiceImpl** extends class **UnicastRemoteObject** (package `java.rmi.server`) and implements **Remote** interface **WeatherService** (lines 13–14). Class **UnicastRemoteObject** provides the basic functionality required for all remote objects. In particular, its constructor *exports* the object to make it available to receive remote calls. Exporting the object enables the remote object to wait for client connections on an *anonymous port number* (i.e., one chosen by the computer on which the remote object executes). This enables the object to perform *unicast communication* (point-
to-point communication between two objects via method calls) using standard streams-based socket connections. RMI abstracts away these communication details so the programmer can work with simple method calls. The `WeatherServiceImpl` constructor (lines 19–23) invokes the default constructor for class `UnicastRemoteObject` (line 21) and calls `private` method `updateWeatherConditions` (line 22). Overloaded constructors for class `UnicastRemoteObject` allow the programmer to specify additional information, such as an explicit port number on which to export the remote object. All `UnicastRemoteObject` constructors throw `RemoteExceptions`.

**Software Engineering Observation 13.5**

Class `UnicastRemoteObject` constructors and methods throw checked `RemoteExceptions`, so `UnicastRemoteObject` subclasses must define constructors that also throw `RemoteExceptions`.

**Software Engineering Observation 13.6**

Class `UnicastRemoteObject` provides basic functionality that remote objects require to handle remote requests. Remote object classes need not extend this class if those remote object classes use `static` method `exportObject` of class `UnicastRemoteObject` to export remote objects.

Method `updateWeatherConditions` (lines 26–91) reads weather information from the `Traveler’s Forecast` Web page and stores this information in a `List` of `WeatherBean` objects. Lines 32–33 create a `URL` object for the `Traveler’s Forecast` Web page. Lines 36–37 invoke method `openStream` of class `URL` to open a connection to the specified `URL` and wrap that connection with a `BufferedReader`.

Lines 40–87 perform HTML scraping (i.e., extracting data from a Web page) to retrieve the weather forecast information. Line 40 defines a separator `String`—"TAV12"—that determines the starting point from which to locate the appropriate weather information. Lines 43–44 read through the `Traveler’s Forecast` Web page until reaching the sentinel. This process skips over information not needed for this application.

Lines 48–51 define two `Strings` that represent the column heads for the weather information. Depending on the time of day, the column headers are either

"CITY WEA HI/LO WEA HI/LO"

after the morning update (normally around 10:30 AM Eastern Standard Time) or

"CITY WEA LO/HI WEA LO/HI"

after the evening update (normally around 10:30 PM Eastern Standard Time).

Lines 65–85 read each city’s weather information and place this information in `WeatherBean` objects. Each `WeatherBean` contains the city’s name, the temperature and a description of the weather. Line 61 creates a `List` for storing the `WeatherBean` objects. Lines 76–79 construct a `WeatherBean` object for the current city. The first 16 characters of `inputLine` are the city name, the next 6 characters of `inputLine` are the description (i.e., weather forecast) and the next 6 characters of `inputLine` are the high and low temperatures. The last two columns of data represent the next day’s weather forecast, which we ignore in this example. Line 82 adds the `WeatherBean` object to the `List`. Line 87 closes the `BufferedReader` and its associated `InputStream`.

Method `getWeatherInformation` (lines 107–110) is the method from interface `WeatherService` that `WeatherServiceImpl` must implement to respond to remote
requests. The method returns a serialized copy of the `weatherInformation List`. Clients invoke this remote method to obtain the weather information.

Method `main` (lines 113–127) creates the `WeatherServiceImpl` remote object. When the constructor executes, it exports the remote object so the object can listen for remote requests. Line 106 defines the URL that a client can use to obtain a `remote reference` to the server object. The client uses this remote reference to invoke methods on the remote object. The URL normally is of the form

```
rmi://host:port/remoteObjectName
```

where `host` represents the computer that is running the `registry for remote objects` (this also is the computer on which the remote object executes), `port` represents the port number on which the registry is running on the `host` and `remoteObjectName` is the name the client will supply when it attempts to locate the remote object in the registry. The `rmiregistry` utility program manages the registry for remote objects and is part of the J2SE SDK. The default port number for the RMI registry is `1099`.

**Software Engineering Observation 13.7**
RMI clients assume that they should connect to port `1099` when attempting to locate a remote object through the RMI registry (unless specified otherwise with an explicit port number in the URL for the remote object).

**Software Engineering Observation 13.8**
A client must specify a port number only if the RMI registry is running on a port other than the default port, `1099`.

In this program, the remote object URL is

```
rmi://localhost/WeatherService
```

indicating that the RMI registry is running on the `localhost` (i.e., the local computer) and that the name the client must use to locate the service is `WeatherService`. The name `localhost` is synonymous with the IP address `127.0.0.1`, so the preceding URL is equivalent to

```
rmi://127.0.0.1/WeatherService
```

Line 124 invokes `static method rebind` of class `Naming` (package `java.rmi`) to bind the remote `WeatherServiceImpl` object `service` to the RMI registry with the URL `rmi://localhost/WeatherService`. There also is a `bind` method for binding a remote object to the registry. Programmers use method `rebind` more commonly, because method `rebind` guarantees that if an object already has registered under the given name, the new remote object will replace the previously registered object. This could be important when registering a new version of an existing remote object.

Class `WeatherBean` (Fig. 13.3) stores data that class `WeatherServiceImpl` retrieves from the National Weather Service Web site. This class stores the city, temperature and weather descriptions as `Strings`. Lines 64–85 provide `get` methods for each piece of information. Lines 25–45 load a property file that contains image names for displaying the weather information. This `static` block ensures that the image names are available as soon as the virtual machine loads the `WeatherBean` class into memory.
// WeatherBean.java
// WeatherBean maintains weather information for one city.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

public class WeatherBean implements Serializable {

private String cityName; // name of city
private String temperature; // city's temperature
private String description; // weather description
private ImageIcon image; // weather image

private static Properties imageNames;

// initialize imageNames when class WeatherBean
// is loaded into memory
static {
    imageNames = new Properties(); // create properties table

    // load weather descriptions and image names from
    // properties file
    try {
        // obtain URL for properties file
        URL url = WeatherBean.class.getResource(
            "imagenames.properties" );

        // load properties file contents
        imageNames.load( new FileInputStream( url.getFile() ) );
    }
    // process exceptions from opening file
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }
} // end static block

// WeatherBean constructor
public WeatherBean( String city, String weatherDescription,
    String cityTemperature )
{
    cityName = city;
    temperature = cityTemperature;
    description = weatherDescription.trim();
}

Fig. 13.3 WeatherBean stores weather forecast for one city (part 1 of 2).
Next, we define the client application that will obtain weather information from the `WeatherServiceImpl`. Class `WeatherServiceClient` (Fig. 13.4) is the client application that invokes remote method `getWeatherInformation` of interface `WeatherService` to obtain weather information through RMI. Class `WeatherServiceClient` uses a `JList` with a custom `ListCellRenderer` to display the weather information for each city.

The `WeatherServiceClient` constructor (lines 16–58) takes as an argument the name of computer on which the `WeatherService` remote object is running. Line 24 creates a `String` that contains the URL for this remote object. Lines 27–28 invoke `Naming`'s static method `lookup` to obtain a remote reference to the `WeatherService` remote object at the specified URL. Method `lookup` connects to the RMI registry and returns a `Remote` reference to the remote object, so line 28 casts this reference to type `WeatherService`. Note that the `WeatherServiceClient` refers to the remote object only through interface `WeatherService`—the remote interface for the `WeatherServiceImpl` remote object implementation. The client can use this remote reference as if it referred to a local object running in the same virtual machine. This remote reference...
refers to a *stub* object on the client. Stubs allow clients to invoke remote objects’ methods. Stub objects receive each remote method call and pass those calls to the RMI system, which performs the networking that allows clients to interact with the remote object. In this case, the `WeatherServiceImpl` stub will handle the communication between `WeatherServiceImpl` and `WeatherServiceClient`. The RMI layer is responsible for network connections to the remote object, so referencing remote objects is transparent to the client. RMI handles the underlying communication with the remote object and the transfer of arguments and return values between the objects.

Lines 31–32 invoke remote method `getWeatherInformation` on the `weatherService` remote reference. This method call returns a copy of the `List` of `WeatherBeans`, which contains information from the *Traveler’s Forecast* Web page. It is important to note that RMI returns a copy of the `List`, because returning a reference from a remote method call is different from returning a reference from a local method call. RMI uses object serialization to send the `List` of `WeatherBean` objects to the client. Therefore, the argument and return types for remote methods must be `Serializable`.

```java
// WeatherServiceClient.java
package com.deitel.advjhtp1.rmi.weather;

import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.List;

import javax.swing.JFrame;

public class WeatherServiceImpl implements WeatherService {
    ...
}

// WeatherServiceClient.java
package com.deitel.advjhtp1.rmi.weather;

import java.rmi.*;
import java.util.*;

import javax.swing.*;

public class WeatherServiceClient extends JFrame {
    // WeatherServiceClient constructor
    public WeatherServiceClient( String server )
    {
        super( "RMI WeatherService Client" );

        try {
            // name of remote server object bound to rmi registry
            String remoteName = "rmi://" + server + "/WeatherService";

            // lookup WeatherServiceImpl remote object
            WeatherService weatherService =
                (WeatherService) Naming.lookup( remoteName );

            // get weather information from server
            List weatherInformation =
                weatherService.getWeatherInformation();
        }
    }
}
```

**Fig. 13.4** `WeatherServiceClient` client for `WeatherService` remote object (part 1 of 2).
Lines 35–36 create a `WeatherListModel` (Fig. 13.5) to facilitate displaying the weather information in a `JList` (line 39). Line 40 sets a `ListCellRenderer` for the `JList`. Class `WeatherCellRenderer` (Fig. 13.6) is a `ListCellRenderer` that uses `WeatherItem` objects to display weather information stored in `WeatherBeans`. 

Fig. 13.4 WeatherServiceClient client for WeatherService remote object (part 2 of 2).
Method `main` (lines 61–77) checks the command-line arguments for a user-provided hostname. If the user did not provide a hostname, line 68 creates a new `WeatherServiceClient` that connects to an RMI registry running on `localhost`. If the user did provide a hostname, line 70 creates a `WeatherServiceClient` using the given hostname.

Class `WeatherListModel` (Fig. 13.5) is a `ListModel` that contains `WeatherBeans` to be displayed in a `JList`. This example continues our design patterns discussion by introducing the `Adapter design pattern`, which enables two objects with incompatible interfaces to communicate with each other. The Adapter design pattern has many parallels in the real world. For example, the electrical plugs on appliances in the United States are not compatible with European electrical sockets. Using an American electrical appliance in Europe requires the user to place an adapter between the electrical plug and the electrical socket. On one side, this adapter provides an interface compatible with the American electrical plug. On the other side, this adapter provides an interface compatible with the European electrical socket. Class `WeatherListModel` plays the role of the `Adapter` in the Adapter design pattern. In Java, interface `List` is not compatible with class `JList`’s interface—a `JList` can retrieve elements only from a `ListModel`. Therefore, we provide class `WeatherListModel`, which adapts interface `List` to make it compatible with `JList`’s interface. When the `JList` invokes `WeatherListModel` method `getSize`, `WeatherListModel method `size` of interface `List`. When the `JList` invokes `WeatherListModel` method `getElementAt`, `WeatherListModel invokes `JList` method `get`, etc. Class `WeatherListModel` also plays the role of the model in Swing’s delegate-model architecture, as we discussed in Chapter 3, Model-View-Controller.

```
// WeatherListModel.java
// WeatherListModel extends AbstractListModel to provide a
// ListModel for storing a List of WeatherBeans.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.util.*;

// Java extension packages
import javax.swing.AbstractListModel;

public class WeatherListModel extends AbstractListModel {

    // List of elements in ListModel
    private List list;

    // no-argument WeatherListModel constructor
    public WeatherListModel() {
        // create new List for WeatherBeans
        list = new ArrayList();
    }
}
```

Fig. 13.5 `WeatherListModel` is a `ListModel` implementation for storing weather information (part 1 of 2).

Fig. 13.5  WeatherListModel is a ListModel implementation for storing weather information (part 2 of 2).
Class `JList` uses a `ListCellRenderer` to render each element in that `JList`’s `ListModel`. Class `WeatherCellRenderer` (Fig. 13.6) is a `DefaultListCellRenderer` subclass for rendering `WeatherBeans` in a `JList`. Method `getListCellRendererComponent` creates and returns a `WeatherItem` (Fig. 13.7) for the given `WeatherBean`.

Class `WeatherItem` (Fig. 13.7) is a `JPanel` subclass for displaying weather information stored in a `WeatherBean`. Class `WeatherCellRenderer` uses instances of class `WeatherItem` to display weather information in a `JList`. The static block (lines 22–29) loads the `ImageIcon backgroundImage` into memory when the virtual machine loads the `WeatherItem` class itself. This ensures that `backgroundImage` is available to all instances of class `WeatherItem`. Method `paintComponent` (lines 38–56) draws the `backgroundImage` (line 43), the city name (line 50), the temperature (line 51) and the `WeatherBean`’s `ImageIcon`, which describes the weather conditions (line 54).

```
1 // WeatherCellRenderer.java
2 // WeatherCellRenderer is a custom ListCellRenderer for
3 // WeatherBeans in a JList.
4 package com.deitel.advjhtp1.rmi.weather;
5
6 // Java core packages
7 import java.awt.*;
8
9 // Java extension packages
10 import javax.swing.*;
11
12 public class WeatherCellRenderer extends DefaultListCellRenderer {
13
14 // returns a WeatherItem object that displays city's weather
15 public Component getListCellRendererComponent( JList list,
16 Object value, int index, boolean isSelected, boolean focus )
17 {
18     return new WeatherItem( ( WeatherBean ) value );
19 }
20 }
```

Fig. 13.6 WeatherCellRenderer is a custom ListCellRenderer for displaying WeatherBeans in a JList.

```
1 // WeatherItem.java
2 // WeatherItem displays a city’s weather information in a JPanel.
3 package com.deitel.advjhtp1.rmi.weather;
4
5 // Java core packages
6 import java.awt.*;
7 import java.net.*;
8 import java.util.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
```

Fig. 13.7 WeatherItem displays weather information for one city (part 1 of 2).
public class WeatherItem extends JPanel {
  private WeatherBean weatherBean;  // weather information

  // background ImageIcon
  private static ImageIcon backgroundImage;

  // static initializer block loads background image when class // WeatherItem is loaded into memory
  static {
    // get URL for background image
    URL url = WeatherItem.class.getResource( "images/back.jpg" );
    backgroundImage = new ImageIcon( url );
  }

  // initialize a WeatherItem
  public WeatherItem( WeatherBean bean )
  {
    weatherBean = bean;
  }

  // display information for city's weather
  public void paintComponent( Graphics g )
  {
    super.paintComponent( g );

    // draw background
    backgroundImage.paintIcon( this, g, 0, 0 );

    // set font and drawing color,
    // then display city name and temperature
    Font font = new Font( "SansSerif", Font.BOLD, 12 );
    g.setFont( font );
    g.setColor( Color.white );
    g.drawString( weatherBean.getCityName(), 10, 19 );
    g.drawString( weatherBean.getTemperature(), 130, 19 );

    // display weather image
    weatherBean.getImage().paintIcon( this, g, 253, 1 );
  } // end method paintComponent

  // make WeatherItem's preferred size the width and height of // the background image
  public Dimension getPreferredSize()
  {
    return new Dimension( backgroundImage.getIconWidth(),
                         backgroundImage.getIconHeight() );
  }
}

Fig. 13.7 WeatherItem displays weather information for one city (part 2 of 2).
13.5 Compiling and Executing the Server and the Client

Now that the pieces are in place, we can build and execute our distributed application; this requires several steps. First, we must compile the classes. Next, we must compile the remote object class (WeatherServiceImpl), using the rmic compiler (a utility supplied with the J2SE SDK) to produce a stub class. As we discussed in Section 13.4, a stub class forwards method invocations to the RMI layer, which performs the network communication necessary to invoke the method call on the remote object. The command line

```
rmic -v1.2 com.deitel.advjhtp1.rmi.weather.WeatherServiceImpl
```

generates the file WeatherServiceImpl_Stub.class. This class must be available to the client (either locally or via download) to enable remote communication with the server object. Depending on the command line options passed to rmic, this may generate several files. In Java 1.1, rmic produced two classes—a stub class and a skeleton class. Java 2 no longer requires the skeleton class. The command-line option -v1.2 indicates that rmic should create only the stub class.

The next step is to start the RMI registry with which the WeatherServiceImpl object will register. The command line

```
rmiregistry
```

launches the RMI registry on the local machine. The command line window (Fig. 13.8) will not show any text in response to this command.

**Common Programming Error 13.1**

Not starting the RMI registry before attempting to bind the remote object to the registry results in a java.rmi.ConnectException, which indicates that the program cannot connect to the registry.

To make the remote object available to receive remote method calls, we bind the object to a name in the RMI registry. Run the WeatherServiceImpl application from the command line as follows:

```
java com.deitel.advjhtp1.rmi.weather.WeatherServiceImpl
```

Figure 13.9 shows the WeatherServiceImpl application output. Class WeatherServiceImpl retrieves the data from the Traveler’s Forecast Web page and displays a message indicating that the service is running.
The `WeatherServiceClient` program now can connect with the `WeatherServiceImpl` running on localhost with the command

```java
java com.deitel.advjhtp1.rmi.weather.WeatherServiceClient
```

Figure 13.10 shows the `WeatherServiceClient` application window. When the program executes, the `WeatherServiceClient` connects to the remote server object and displays the current weather information.

If the `WeatherServiceImpl` is running on a different machine from the client, you can specify the IP address or host name of the server computer as a command-line argument when executing the client. For example, to access a computer with IP address `192.168.0.150`, enter the command

```java
java com.deitel.advjhtp1.rmi.weather.WeatherServiceClient
192.168.0.150
```

In the first part of this chapter, we built a simple distributed system that demonstrated the basics of RMI. In the following case study, we build a more sophisticated RMI distributed system that takes advantage of some advanced RMI features.
13.6 Case Study: Deitel Messenger with Activatable Server

In this section, we present a case study that implements an online chat system using RMI and an *activatable* chat server. This case study—the Deitel Messenger—uses several advanced RMI features and a modular architecture that promotes reusability. Figure 13.11 lists the classes and interfaces that make up the case study and brief descriptions of each. Interfaces are shown in italic font.

Standard RMI objects exported as `UnicastRemoteObject` must run continuously on the server to handle client requests. RMI objects that extend class `java.rmi.activation.Activatable` are able to *activate*, or start running, when a client invokes one of the remote object’s methods. This can conserve resources on the server because a remote object’s processes are put to sleep and release memory when there are no clients using that particular remote object. The *RMI activation daemon* (`rmid`) is a server process that enables activatable remote objects to become active when clients invoke remote methods on these objects.

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChatServer</td>
<td>Remote interface through which clients register for a chat, leave a chat, and post chat messages.</td>
</tr>
<tr>
<td>StoppableChatServer</td>
<td>Administrative remote interface for terminating the chat server.</td>
</tr>
<tr>
<td>ChatServerImpl</td>
<td>Implementation of the ChatServer remote interface that provides an RMI-based chat server.</td>
</tr>
<tr>
<td>ChatServerAdministrator</td>
<td>Utility program for launching and terminating the activatable ChatServer.</td>
</tr>
<tr>
<td>ChatClient</td>
<td>Remote interface through which the ChatServer communicates with clients.</td>
</tr>
<tr>
<td>ChatMessage</td>
<td><code>Serializable</code> object for sending messages between ChatServer and ChatClients.</td>
</tr>
<tr>
<td>MessageManager</td>
<td>Interface that defines methods for managing communication between the client’s user interface and the ChatServer.</td>
</tr>
<tr>
<td>RMIMessageManager</td>
<td>ChatClient and MessageManager implementation for managing communication between the client and the ChatServer.</td>
</tr>
<tr>
<td>MessageListener</td>
<td>Interface for classes that wish to receive new chat messages.</td>
</tr>
<tr>
<td>DisconnectListener</td>
<td>Interface for classes that wish to receive notifications when the server disconnects.</td>
</tr>
<tr>
<td>ClientGUI</td>
<td>GUI for sending and receiving chat messages using a MessageManager.</td>
</tr>
<tr>
<td>DeitelMessenger</td>
<td>Application launcher for Deitel Messenger client.</td>
</tr>
</tbody>
</table>

Fig. 13.11 Participants of DeitelMessenger case study.
Activatable remote objects also are able to recover from server crashes, because remote references to activatable objects are persistent—when the server restarts, the RMI activation daemon maintains the remote reference, so clients can continue to use the remote object. We discuss the details of implementing Activatable remote objects when we present the chat server implementation.

### 13.6.1 Activatable Deitel Messenger ChatServer

Like every RMI remote object, an Activatable remote object must implement a remote interface. Interface **ChatServer** (Fig. 13.12) is the remote interface for the Deitel Messenger server. Clients interact with the Deitel Messenger server through the ChatServer remote interface. Remote interfaces for an Activatable RMI have the same requirements as standard RMI remote interfaces.

Line 13 declares that interface **ChatServer** extends interface **Remote**, which RMI requires for all remote interfaces. Method **registerClient** (lines 16–17) enables a ChatClient (Fig. 13.17) to register with the ChatServer and take part in the chat session. Method **registerClient** takes as an argument the ChatClient to register. Interface **ChatClient** is itself a remote interface, so both the server and client are remote objects in this application. This enables the server to communicate with clients by invoking remote methods on those clients. We discuss this communication—called an RMI callback—in more detail when we present the ChatClient implementation.

```java
public interface ChatServer extends Remote {

    // register new ChatClient with ChatServer
    public void registerClient( ChatClient client )
    throws RemoteException;

    // unregister ChatClient with ChatServer
    public void unregisterClient( ChatClient client )
    throws RemoteException;

    // post new message to ChatServer
    public void postMessage( ChatMessage message )
    throws RemoteException;
}
```

**Fig. 13.12** **ChatServer** remote interface for Deitel Messenger chat server.
Method `unregisterClient` (lines 20–21) enables clients to remove themselves from the chat session. Method `postMessage` enables clients to post new messages to the chat session. Method `postMessage` takes as an argument a reference to a `ChatMessage`. A `ChatMessage` (Fig. 13.18) is a `Serializable` object that contains the name of the sender and the message body. We discuss this class in more detail shortly.

The server side of the Deitel Messenger application includes a program for managing the `Activatable` remote object. Interface `StoppableChatServer` (Fig. 13.13) declares method `stopServer`. The program that manages the Deitel Messenger server invokes method `stopServer` to terminate the server.

Class `ChatServerImpl` (Fig. 13.14) is an `Activatable` RMI object that implements the `ChatServer` and `StoppableChatServer` remote interfaces. Line 23 creates a `Set` for maintaining remote references to registered `ChatClient`s. The `ChatServerImpl` constructor (lines 29–34) takes as arguments an `ActivationID` and a `MarshalledObject`. The RMI activation mechanism requires that `Activatable` objects provide this constructor. When the activation daemon activates a remote object of this class, it invokes this `activation constructor`. The `ActivationID` argument specifies a unique identifier for the remote object. Class `MarshalledObject` is a wrapper class that contains a serialized object for transmission over RMI. In this case, the `MarshalledObject` argument contains application-specific initialization information, such as the name under which the activation daemon registered the remote object. Line 33 invokes the superclass constructor to complete activation. The second argument to the superclass constructor (0) specifies that the activation daemon should export the object on an anonymous port.

```java
// StoppableChatServer.java
// StoppableChatServer is a remote interface that provides a
// mechanism to terminate the chat server.
package com.deitel.messenger.rmi.server;

// Java core packages
import java.rmi.*;

public interface StoppableChatServer extends Remote {

    // stop ChatServer
    public void stopServer() throws RemoteException;
}
```

Fig. 13.13 `StoppableChatServer` remote interface for stopping a `ChatServer` remote object.

```java
// ChatServerImpl.java
// ChatServerImpl implements the ChatServer remote interface
// to provide an RMI-based chat server.
package com.deitel.messenger.rmi.server;

Fig. 13.14 `ChatServerImpl` implementation of remote interfaces `ChatServer` and `StoppableChatServer` as `Activatable` remote objects (part 1 of 5).
```
// Java core packages
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

// Deitel packages
import com.deitel.messenger.rmi.ChatMessage;
import com.deitel.messenger.rmi.client.ChatClient;

public class ChatServerImpl extends Activatable
    implements ChatServer, StoppableChatServer {

    // Set of ChatClient references
    private Set clients = new HashSet();

    // server object's name
    private String serverObjectName;

    // ChatServerImpl constructor
    public ChatServerImpl( ActivationID id, MarshalledObject data )
        throws RemoteException {
        super( id, 0 );
    }

    // register ChatServerImpl object with RMI registry.
    public void register( String rmiName )
        throws RemoteException,
        IllegalArgumentException, MalformedURLException
    {
        // ensure registration name was provided
        if ( rmiName == null )
            throw new IllegalArgumentException(
                "Registration name cannot be null" );

        serverObjectName = rmiName;

        // bind ChatServerImpl object to RMI registry
        try {
            // create RMI registry
            System.out.println( "Creating registry ..." );
            Registry registry =
                LocateRegistry.createRegistry( 1099 );

            // bind RMI object to default RMI registry
            System.out.println( "Binding server to registry ..." );

            // Register ChatServerImpl object
            registry bind new ChatServerImpl( rmiName, data );
        }
    }

Fig. 13.14 ChatServerImpl implementation of remote interfaces ChatServer
and StoppableChatServer as Activatable remote objects (part
2 of 5).
registry.rebind( serverObjectName, this );

// if registry already exists, bind to existing registry
catch ( RemoteException remoteException ) {
    System.err.println( "Registry already exists. " +
            "Binding to existing registry ..." );
    Naming.rebind( serverObjectName, this );
}

System.out.println( "Server bound to registry" );

} // end method register

// register new ChatClient with ChatServer
public void registerClient( ChatClient client )
    throws RemoteException
{
    // add client to Set of registered clients
    synchronized ( clients ) {
        clients.add( client );
    }

    System.out.println( "Registered Client: " + client );

} // end method registerClient

// unregister client with ChatServer
public void unregisterClient( ChatClient client )
    throws RemoteException
{
    // remove client from Set of registered clients
    synchronized( clients ) {
        clients.remove( client );
    }

    System.out.println( "Unregistered Client: " + client );

} // end method unregisterClient

// post new message to chat server
public void postMessage( ChatMessage message )
    throws RemoteException
{
    Iterator iterator = null;

    // get Iterator for Set of registered clients
    synchronized( clients ) {
        iterator = new HashSet( clients ).iterator();
    }

    // ...
// send message to every ChatClient
while ( iterator.hasNext() ) {

    // attempt to send message to client
    ChatClient client = ( ChatClient ) iterator.next();

    try {
        client.deliverMessage( message );
    }

    // unregister client if exception is thrown
    catch( Exception exception ) {
        System.err.println( "Unregistering absent client." );
        unregisterClient( client );
    }
}

} // end while loop

} // end method postMessage

// notify each client that server is shutting down and
// terminate server application
public void stopServer() throws RemoteException {

    System.out.println( "Terminating server ..." );

    Iterator iterator = null;

    // get Iterator for Set of registered clients
    synchronized( clients ) {
        iterator = new HashSet( clients ).iterator();
    }

    // send message to every ChatClient
    while ( iterator.hasNext() ) {
        ChatClient client = ( ChatClient ) iterator.next();
        client.serverStopping();
    }

    // create Thread to terminate application after
    // stopServer method returns to caller
    Thread terminator = new Thread( new Runnable() {

        // sleep for 5 seconds, print message and terminate
        public void run()
        {
            // sleep
            try {
                Thread.sleep( 5000 );
            }

        }
    });

Method `register` (lines 37–69) registers a `ChatServerImpl` remote object with the RMI registry. If the provided name for the remote object is `null`, lines 42–43 throw an `IllegalArgumentException`, indicating that the caller must specify a name for the remote object. Lines 52–53 use `static` method `createRegistry` of class `LocalRegistry` to create a new `Registry` on the local machine at port `1099`, which is the default port. This is equivalent to executing the `rmiregistry` utility to start a new RMI registry. Line 57 invokes method `rebind` of class `Registry` to bind the activatable object to the `Registry`. If creating or binding to the `Registry` fails, we assume that an RMI registry already is running on the local machine. Line 64 invokes `static` method `rebind` of class `Naming` to bind the remote object to the existing RMI registry.

Method `registerClient` (lines 72–82) enables `ChatClient` remote objects to register with the `ChatServer` to participate in the chat session. The `ChatClient` argument to method `registerClient` is a remote reference to the registering client, which is itself a remote object. Line 77 adds the `ChatClient` remote reference to the `Set` of `ChatClient`s participating in the chat session. Method `unregisterClient` (lines 85–95) enables `ChatClients` to leave the chat session. Line 90 removes the given `ChatClient` remote reference from the `Set` of `ChatClient` references.

`ChatClients` invoke method `postMessage` (lines 98–124) to post new `ChatMessages` to the chat session. Each `ChatMessage` (Fig. 13.18) instance is a `Serializable` object that contains as properties the message sender and the message body. Lines 109–123 iterate through the `Set` of `ChatClient` references and invoke remote method `deliverMessage` of interface `ChatClient` to deliver the new `ChatMessage` to each client. If delivering a message to a client throws an exception, we assume that the client is no longer available. Line 121 therefore unregisters the absent client from the server.

Interface `StoppableChatServer` requires that class `ChatServerImpl` implements method `stopServer` (lines 128–170). Lines 140–143 iterate through the `Set` of `ChatClient` references and invoke method `serverStopping` of interface `ChatClient` to notify each `ChatClient` that the server is shutting down. Lines 147–168
create and start a new Thread to ensure that the ChatServerAdministrator (Fig. 13.15) can unbind the remote object from the RMI Registry before the remote object terminates.

Class ChatServerAdministrator (Fig. 13.15) is a utility program for registering and unregistering the activatable ChatServer remote object. Method startServer (lines 14–52) launches the activatable ChatServer. Activatable RMI objects execute as part of an ActivationGroup (package java.rmi.activation). The RMI activation daemon starts a new virtual machine for each ActivationGroup. Lines 21–22 create a Properties object and add a property that specifies the policy file under which the ActivationGroup’s JVM should run. This policy file (Fig. 13.16) allows Activatable objects in this ActivationGroup to terminate the virtual machine for this activation group. Recall that ChatServerImpl invokes static method exit of class System in method stopServer, which terminates the ActivationGroup’s virtual machine along with all of its executing remote objects.

```
// ChatServerAdministrator.java
// ChatServerAdministrator is a utility program for launching and terminating the Activatable ChatServer.
package com.deitel.messenger.rmi.server;

// Java core packages
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

public class ChatServerAdministrator {

    // set up activatable server object
    private static void startServer( String policy, String codebase ) throws Exception {
        // set up RMI security manager
        System.setSecurityManager( new RMISecurityManager() );

        // set security policy for ActivatableGroup JVM
        Properties properties = new Properties();
        properties.put( "java.security.policy", policy );

        // create ActivationGroupDesc for activatable object
        ActivationGroupDesc groupDesc =
            new ActivationGroupDesc( properties, null );

        // register activation group with RMI activation system
        ActivationGroupID groupID =
            ActivationGroup.getSystem().registerGroup( groupDesc );

        // create activation group
        ActivationGroup.createGroup( groupID, groupDesc , 0 );
    }
```

Fig. 13.15 ChatServerAdministrator application for starting and stopping the ChatServer remote object (part 1 of 3).
// activation description for ChatServerImpl
ActivationDesc description = new ActivationDesc(
    "com.deitel.messenger.rmi.server.ChatServerImpl",
    codebase, null );

// register description with rmid
ChatServer server =
    ( ChatServer ) Activatable.register( description );
System.out.println( "Obtained ChatServerImpl stub" );

// bind ChatServer in registry
Naming.rebind( "ChatServer", server );
System.out.println( "Bound object to registry" );

// terminate setup program
System.exit( 0 );

} // end method startServer

// terminate server
private static void terminateServer( String hostname )
    throws Exception
{
    // lookup ChatServer in RMI registry
    System.out.println( "Locating server ..." );
    StoppableChatServer server = ( StoppableChatServer )
        Naming.lookup( "rmi://" + hostname + "/ChatServer" );

    // terminate server
    System.out.println( "Stopping server ..." );
    server.stopServer();

    // remove ChatServer from RMI registry
    System.out.println( "Server stopped" );
    Naming.unbind( "rmi://" + hostname + "/ChatServer" );

} // end method terminateServer

// launch ChatServerAdministrator application
public static void main( String args[] ) throws Exception
{
    // check for stop server argument
    if ( args.length == 2 ) {
        if ( args[ 0 ].equals( "stop" ) )
            terminateServer( args[ 1 ] );
    else printUsageInstructions();
    }

Fig. 13.15 ChatServerAdministrator application for starting and stopping the ChatServer remote object (part 2 of 3).
Lines 25–26 create an `ActivationGroupDesc` object, which is an *activation group descriptor*. The activation group descriptor specifies configuration information for the `ActivationGroup`. The first argument to the `ActivationGroupDesc` constructor is a `Properties` reference that contains replacement values for system properties in the `ActivationGroup`'s virtual machine. In this example, we override the `java.security.policy` system property to provide an appropriate security policy for the `ActivationGroup`'s virtual machine. The second argument is a reference to an `ActivationGroupDesc.CommandEnvironment` object. This object enables the `ActivationGroup` to customize the commands that the activation daemon executes when starting the `ActivationGroup`'s virtual machine. This example requires no such customization, so we pass a `null` reference for the second argument.

Lines 29–30 obtain an `ActivationSystem` by invoking static method `getSystem` of class `ActivationGroup`. Line 30 invokes method `registerGroup` of interface `ActivationSystem` and passes as an argument the `groupDesc` activation
group descriptor. Method `registerGroup` returns the `ActivationGroupID` for the newly registered `ActivationGroup`. Line 33 invokes static method `createGroup` of class `ActivationGroup` to create the `ActivationGroup`. This method takes as arguments the `ActivationGroupID`, the `ActivationGroupDesc` and the incarnation number for the `ActivationGroup`. The incarnation number identifies different instances of the same `ActivationGroup`. Each time the activation daemon activates the `ActivationGroup`, the daemon increments the incarnation number.

Lines 36–38 create an `ActivationDesc` object for the `ChatServer` remote object. This activation descriptor specifies configuration information for a particular `Activatable` remote object. The first argument to the `ActivationDesc` constructor specifies the name of the class that implements the `Activatable` remote object. The second argument specifies the codebase that contains the remote object’s class files. The final argument is a `MarshalledObject` reference, whose object specifies initialization information for the remote object. Recall that the `ChatServerImpl` activation constructor takes as its second argument a `MarshalledObject` reference. Our `ChatServer` remote object requires no special initialization information, so line 38 passes a null reference for the `MarshalledObject` argument.

Line 42 invokes static method `register` of class `Activatable` to register the `Activatable` remote object. Method `register` takes as an argument the `ActivationDesc` for the `Activatable` object and returns a reference to the remote object’s stub. Line 46 invokes static method `rebind` of class `Naming` to bind the `ChatServer` in the RMI Registry.

Method `terminateServer` (lines 55–71) provides a means to shut down the activatable `ChatServer` remote object. Line 61 invokes static method `lookup` of class `Naming` to obtain a remote reference to the `ChatServer`. Line 60 casts the reference to type `StoppableChatServer`, which declares method `stopServer`. Line 65 invokes method `stopServer` to notify clients that the `ChatServer` is shutting down. Recall that method `stopServer` of class `ChatServerImpl` starts a Thread that waits five seconds before invoking static method `exit` of class `System`. This Thread keeps the `ChatServer` remote object running after method `stopServer` returns, allowing the `ChatServerAdministrator` to remove the remote object from the RMI Registry. Line 69 invokes static method `unbind` of class `Naming` to remove the `ChatServer` remote object from the RMI Registry. The Thread in class `ChatServerImpl` then terminates the virtual machine in which the `ChatServer` remote object ran.

Method `main` (lines 74–99) checks the command-line arguments to determine whether to start or stop the `ChatServer` remote object. When stopping the server, the user must provide as the second argument the hostname of the computer on which the server is running. When starting the server, the user must provide as arguments the location of the policy file for the `ActivationGroup` and the codebase for the remote object. If the user passes argument "stop", line 80 invokes method `terminateServer` to shut down the `ChatServer` on the specified host. If the user passes argument "start", line 91 invokes method `startServer` with the given policy file location and codebase. If the user provides an invalid number or type of arguments, lines 82, 93 and 97 invoke method `printUsageInstructions` (lines 102–109) to display information about the required command-line arguments.
13.6.2 Deitel Messenger Client Architecture and Implementation

Throughout this book, we present several versions of the Deitel Messenger case study. Each version implements the underlying communications using a different technology. For example, in Chapter 26, Common Object Request Broker Architecture (CORBA): Part 1, we present an implementation that uses CORBA as the underlying communication mechanism. The client for the Deitel Messenger application uses a modularized architecture to optimize code reuse in the several versions of this case study.

Communication Interfaces and Implementation

The client for the Deitel Messenger system separates the application GUI and the network communication into separate objects that interact through a set of interfaces. This enables us to use the same client-side GUI for different versions of the Deitel Messenger application. In this section, we present these interfaces and implementations with RMI.

Interface ChatClient (Fig. 13.17) is an RMI remote interface that enables the ChatServer to communicate with the ChatClient through RMI callbacks—remote method calls from the ChatServer back to the client. Recall that when a client connects to the ChatServer, the client invokes ChatServer method registerClient and passes as an argument a ChatClient remote reference. The server then uses this ChatClient remote reference to invoke RMI callbacks on the ChatClient (e.g., to deliver ChatMessages to that client). Method deliverMessage (lines 16–17) enables the ChatServer to send new ChatMessages to the ChatClient. Method serverStopping (line 20) enables the ChatServer to notify the ChatClient when the ChatServer is shutting down.

```java
1 // ChatClient.java
2 // ChatClient is a remote interface that defines methods for a chat client to receive messages and status information from a ChatServer.
3 // a ChatServer.
4 package com.deitel.messenger.rmi.client;
5
6 // Java core packages
7 import java.rmi.*;
8
9 // Deitel packages
10 import com.deitel.messenger.rmi.ChatMessage;
11
12 public interface ChatClient extends Remote {
13     // method called by server to deliver message to client
14     public void deliverMessage( ChatMessage message )
15             throws RemoteException;
16
17     // method called when server shutting down
18     public void serverStopping() throws RemoteException;
19 }
```

Fig. 13.17 ChatClient remote interface to enable RMI callbacks.
Class **ChatMessage** (Fig. 13.18) is a **Serializable** class that represents a message in the Deitel Messenger system. Instance variables `sender` and `message` contain the name of the person who sent the message and the message body, respectively. Class **ChatMessage** provides `set` and `get` methods for the `sender` and `message` and method `toString` for producing a `String` representation of a `ChatMessage`.

Interface **MessageManager** (Fig. 13.19) declares methods for classes that implement communication logic for a **ChatClient**. The methods that this interface declares are not specific to any underlying communication implementation. The chat client GUI uses a **MessageManager** implementation to connect to and disconnect from the **ChatServer**, and to send messages. Method `connect` (lines 10–11) connects to the **ChatServer** and takes as an argument the **MessageListener** to which the **MessageManager** should deliver new incoming messages. We discuss interface **MessageListener** in detail when we present the client user interface. Method `disconnect` (lines 15–16) disconnects the **MessageManager** from the **ChatServer** and stops routing messages to the given **MessageListener**. Method `sendMessage` (lines 19–20) takes as `String` arguments a user name (`from`) and a `message` to send to the **ChatServer**. Method `setDisconnectListener` registers a **DisconnectListener** to be notified when the **ChatServer** disconnects the client. We discuss interface **DisconnectListener** in detail when we present the client user interface.

```java
// ChatMessage.java
// ChatMessage is a Serializable object for messages in the RMI
// ChatClient and ChatServer.
package com.deitel.messenger.rmi;

// Java core packages
import java.io.*;

public class ChatMessage implements Serializable {

    private String sender; // person sending message
    private String message; // message being sent

    // construct empty ChatMessage
    public ChatMessage()
    {
        this( "", "" );
    }

    // construct ChatMessage with sender and message values
    public ChatMessage( String sender, String message )
    {
        setSender( sender );
        setMessage( message );
    }

    // construct ChatMessage with sender and message values
    public ChatMessage( String sender, String message )
    {
        setSender( sender );
        setMessage( message );
    }

Fig. 13.18 ChatMessage is a serializable class for transmitting messages over RMI (part 1 of 2).
```
// set name of person sending message
public void setSender( String name )
{
    sender = name;
}

// get name of person sending message
public String getSender()
{
    return sender;
}

// set message being sent
public void setMessage( String messageBody )
{
    message = messageBody;
}

// get message being sent
public String getMessage()
{
    return message;
}

// String representation of ChatMessage
public String toString()
{
    return getSender() + '> ' + getMessage();
}

Fig. 13.18 ChatMessage is a serializable class for transmitting messages over RMI (part 2 of 2).

// MessageManager.java
// MessageManager is an interface for objects capable of managing
// communications with a message server.
package com.deitel.messenger;

public interface MessageManager {

// connect to message server and route incoming messages
// to given MessageListener
public void connect( MessageListener listener )
    throws Exception;

// disconnect from message server and stop routing
// incoming messages to given MessageListener
public void disconnect( MessageListener listener )
    throws Exception;

Fig. 13.19 MessageManager interface for classes that implement communication logic for a ChatClient (part 1 of 2).
Class \texttt{RMIMessageManager} (Fig. 13.20) handles all communication between the client and the \texttt{ChatServer}. Class \texttt{RMIMessageManager} is an RMI remote object that extends class \texttt{UnicastRemoteObject} and implements the \texttt{ChatClient} remote interface (lines 18–19). Class \texttt{RMIMessageManager} also implements interface \texttt{MessageManager}, enabling the client user interface to use an \texttt{RMIMessageManager} object to communicate with the \texttt{ChatServer}.

The \texttt{RMIMessageManager} constructor takes as a \texttt{String} argument the hostname of the computer running the RMI registry with which the \texttt{ChatServer} has registered. Note that because class \texttt{RMIMessenger} is itself an RMI remote object, the \texttt{RMIMessageManager} constructor throws \texttt{RemoteException}, which RMI requires of all \texttt{UnicastRemoteObject} subclasses. Line 31 assigns the given server name to instance variable \texttt{serverAddress}.

```java
public class RMIMessageManager extends UnicastRemoteObject
    implements ChatClient, MessageManager {

    // listeners for incoming messages and disconnect notifications
    private MessageListener messageListener;
    private DisconnectListener disconnectListener;
```
```java
private String serverAddress;
private ChatServer chatServer;

// RMIMessageManager constructor
public RMIMessageManager( String server ) throws RemoteException {
    serverAddress = server;
}

// connect to ChatServer
public void connect( MessageListener listener ) throws Exception {
    // look up ChatServer remote object
    chatServer = ( ChatServer ) Naming.lookup("" + serverAddress + "/ChatServer");

    // register with ChatServer to receive messages
    chatServer.registerClient( this );

    // set listener for incoming messages
    messageListener = listener;
}

// disconnect from ChatServer
public void disconnect( MessageListener listener ) throws Exception {
    if ( chatServer == null ) return;

    // unregister with ChatServer
    chatServer.unregisterClient( this );

    // remove references to ChatServer and MessageListener
    chatServer = null;
    messageListener = null;
}

// send ChatMessage to ChatServer
public void sendMessage( String fromUser, String message ) throws Exception {
    if ( chatServer == null ) return;

    // create ChatMessage with message text and userName
    ChatMessage chatMessage = new ChatMessage( fromUser, message );
```

**Fig. 13.20** RMIMessageManager remote object and MessageManager implementation for managing ChatClient communication (part 2 of 3).
Method `connect` (lines 35–48)—declared in interface `MessageManager`—connects the `RMIMessageManager` to the `ChatServer`. Lines 39–40 invoke `static` method `lookup` of class `Naming` to retrieve a remote reference to the `ChatServer`. Line 43 invokes method `registerClient` of interface `ChatServer` to register the `RMIMessageManager` for RMI callbacks from the `ChatServer`. Note that line 43 passes the `this` reference as the argument to method `registerClient`. Recall that class `RMIMessageManager` is a remote object, therefore the `this` reference can serve as a remote `ChatClient` reference to the `RMIMessageManager` remote object.

Method `disconnect` (lines 51–64) disconnects the `RMIMessageManager` from the `ChatServer`. If remote `ChatServer` reference `chatServer` is `null`, line 55 returns immediately, because the `RMIMessageManager` is disconnected already. Line 58 invokes method `unregisterClient` of remote interface `ChatServer` to unregister the `RMIMessageManager` from the `ChatServer`. Line 58 passes the `this` reference as an
argument to method `unregisterClient`, specifying that the `ChatServer` should `unregister` this `RMIMessageManager` remote object. Line 62 sets `MessageListener` reference `messageListener` to `null`.

Method `sendMessage` (lines 67–80) delivers a message from the client to the `ChatServer`. Line 71 returns immediately if the `chatServer` remote reference is `null`. Lines 74–75 create a new `ChatMessage` object to contain the user name from whom the message came and the message body. Line 78 invokes method `postMessage` of remote interface `ChatServer` to post the new `ChatMessage` to the `ChatServer`. The `ChatServer` will use RMI callbacks to deliver this message to each registered `ChatClient`.

Method `deliverMessage` (lines 83–89)—defined in remote interface `ChatClient`—enables the `ChatServer` to use RMI callbacks to deliver incoming `ChatMessages` to the `ChatClient`. If there is a `MessageListener` registered with the `RMIMessageManager` (line 86), lines 87–88 invoke method `messageReceived` of interface `MessageListener` to notify the `MessageListener` of the incoming `ChatMessage`. Lines 87–88 invoke methods `getSender` and `getMessage` of class `ChatMessage` to retrieve the message sender and message body, respectively.

Method `serverStopping` (lines 92–96)—defined in remote interface `ChatClient`—enables the `ChatServer` to use RMI callbacks to notify the `ChatClient` that the `ChatServer` is shutting down so the `ChatClient` can disconnect and notify the `DisconnectListener`. Line 95 invokes method `fireServerDisconnected` of class `RMIMessageManager` to notify the registered `DisconnectListener` that the `ChatServer` has disconnected the `ChatClient`.

Method `setDisconnectListener` (lines 99–103)—defined in interface `MessageManager`—enables a `DisconnectListener` to register for notifications when the `ChatServer` disconnects the client. For example, the client user interface could register for these notifications to notify the user that the server has disconnected. Method `fireServerDisconnected` (lines 106–110) is a utility method for sending `serverDisconnected` messages to the `DisconnectListener`. If there is a registered `DisconnectListener`, line 109 invokes method `serverDisconnected` of interface `DisconnectListener` to notify the listener that the server disconnected. We discuss interface `DisconnectListener` in detail when we present the client user interface.

**Client GUI Interfaces and Implementation**

We uncouple the client user interface from the `MessageManager` implementation through interfaces `MessageListener` and `DisconnectListener` (Fig. 13.19 and 13.20). Class `ClientGUI` uses implementations of interfaces `MessageListener` and `DisconnectListener` to interact with the `MessageManager` and provides a graphical user interface for the client.

Interface `MessageListener` (Fig. 13.21) enables objects of an implementing class to receive incoming messages from a `MessageManager`. Line 9 defines method `messageReceived`, which takes as arguments the user name `from` whom the message came and the `message` body.

Interface `DisconnectListener` (Fig. 13.22) enables implementing objects to receive notifications when the server disconnects the `MessageManager`. Line 9 defines method `serverDisconnected`, which takes as a `String` argument a `message` that indicates why the server disconnected.
Chapter 13

Class **ClientGUI** (Fig. 13.23) provides a user interface for the Deitel Messenger client. The GUI consists of a menu and a toolbar with **Actions** for connecting to and disconnecting from a **ChatServer**, a **JTextArea** for displaying incoming **ChatMessages** and a **JTextArea** and **JButton** for sending new messages to the **ChatServer**. Lines 27–29 declare **Action** references for connecting to and disconnecting from the **ChatServer** and for sending **ChatMessages**. Line 35 declares a **MessageManager** reference for the **MessageManager** implementation that provides the network communication. Line 38 declares a **MessageListener** reference for receiving new **ChatMessages** from the **ChatServer** through the **MessageManager**.

```
1 // ClientGUI.java
2 // ClientGUI provides a GUI for sending and receiving
3 // chat messages using a MessageManager.
4 package com.deitel.messenger;
5
6 // Java core packages
7 import java.awt.*;
8 import java.awt.event.*;
9 import java.util.*;
```

Fig. 13.23 **ClientGUI** provides a graphical user interface for the Deitel Messenger client (part 1 of 9).
// Java extension packages
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.text.*;

public class ClientGUI extends JFrame {

    // JLabel for displaying connection status
    private JLabel statusBar;

    // JTextAreas for displaying and inputting messages
    private JTextArea messageArea;
    private JTextArea inputArea;

    // Actions for connecting and disconnecting MessageManager
    // and sending messages
    private Action connectAction;
    private Action disconnectAction;
    private Action sendAction;

    // userName to add to outgoing messages
    private String userName = "";

    // MessageManager for communicating with server
    MessageManager messageManager;

    // MessageListener for receiving new messages
    MessageListener messageListener;

    // ClientGUI constructor
    public ClientGUI( MessageManager manager )
    {
        super( "Deitel Messenger" );
        messageManager = manager;
        messageListener = new MyMessageListener();
        connectAction = new ConnectAction();
        disconnectAction = new DisconnectAction();
        disconnectAction.setEnabled( false );
        sendAction = new SendAction();
        sendAction.setEnabled( false );
        JMenu fileMenu = new JMenu( "File" );
        fileMenu.setMnemonic( 'F' );
        fileMenu.add( connectAction );
        fileMenu.add( disconnectAction );
    }
}

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger client (part 2 of 9).
// set up JMenuBar and attach File menu
JMenuBar menuBar = new JMenuBar();
menuBar.add ( fileMenu );
setJMenuBar( menuBar );

// set up JToolBar
JToolBar toolBar = new JToolBar();
toolBar.add( connectAction );
toolBar.add( disconnectAction );

// create JTextArea for displaying messages
messageArea = new JTextArea( 15, 15 );

// disable editing and wrap words at end of line
messageArea.setEditable( false );
messageArea.setLineWrap( true );
messageArea.setWrapStyleWord( true );

JPanel panel = new JPanel();
panel.setLayout( new BorderLayout( 5, 5 ) );
panel.add( new JScrollPane( messageArea ),
BorderLayout.CENTER );

// create JTextArea for entering new messages
inputArea = new JTextArea( 3, 15 );
inputArea.setLineWrap( true );
inputArea.setWrapStyleWord( true );
inputArea.setEditable( false );

// map Enter key in inputArea to sendAction
Keymap keyMap = inputArea.getKeymap();
KeyStroke enterKey = KeyStroke.getKeyStroke(
    KeyEvent.VK_ENTER, 0 );
keyMap.addActionForKeyStroke( enterKey, sendAction );

// lay out inputArea and sendAction JButton in BoxLayout
// and add Box to messagePanel
Box box = new Box( BoxLayout.X_AXIS );
box.add( new JScrollPane( inputArea ) );
box.add( new JButton( sendAction ) );
panel.add( box, BorderLayout.SOUTH );

// create statusBar JLabel with recessed border
statusBar = new JLabel( "Not Connected" );
statusBar.setBorder( new BevelBorder( BevelBorder.LOWERED ) );

// lay out components
Container container = getContentPane();
container.add( toolBar, BorderLayout.NORTH );
container.add( panel, BorderLayout.CENTER );

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger client (part 3 of 9).
container.add( statusBar, BorderLayout.SOUTH );

// disconnect and exit if user closes window
addWindowListener( 

    new WindowAdapter() { 

        // disconnect MessageManager when window closes
        public void windowClosing( WindowEvent event ) { 

            // disconnect from chat server
            try { 
                messageManager.disconnect( messageListener ); 
            } 

            // handle exception disconnecting from server
            catch ( Exception exception ) { 
                exception.printStackTrace(); 
            } 

            System.exit( 0 );

        } // end method windowClosing

    } // end WindowAdapter inner class

} // end ClientGUI constructor

// Action for connecting to server
private class ConnectAction extends AbstractAction { 

    // configure ConnectAction
    public ConnectAction() {
        putValue( Action.NAME, "Connect" );
        putValue( Action.SMALL_ICON, new ImageIcon( 
            ClientGUI.class.getResource( 
            "images/Connect.gif" ) ) );
        putValue( Action.SHORT_DESCRIPTION, 
            "Connect to Server" );
        putValue( Action.LONG_DESCRIPTION, 
            "Connect to server to send Instant Messages" );
        putValue( Action.MNEMONIC_KEY, new Integer( 'C' ) );
    }

    // connect to server
    public void actionPerformed( ActionEvent event ) {
        try {
            // connect MessageManager to server
            System.exit( 0 );
        } 
    }

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger client (part 4 of 9).
// clear messageArea
messageArea.setText( "" );

// connect MessageManager and register MessageListener
messageManager.connect( messageListener );

// listen for disconnect notifications
messageManager.setDisconnectListener( new DisconnectHandler() );

// get desired userName
userName = JOptionPane.showInputDialog( ClientGUI.this, "Please enter your name: " );

// update Actions, inputArea and statusBar
connectAction.setEnabled( false );
disconnectAction.setEnabled( true );
sendAction.setEnabled( true );
inputArea.setEditable( true );
inputArea.requestFocus();
statusBar.setText( "Connected: " + userName );

// send message indicating user connected
messageManager.sendMessage( userName, userName + " joined chat" );

} // end try

// handle exception connecting to server
try ( Exception exception ) {
    JOptionPane.showMessageDialog( ClientGUI.this, "Unable to connect to server.", "Error Connecting", JOptionPane.ERROR_MESSAGE );
    exception.printStackTrace();
}

} // end method actionPerformed

} // end ConnectAction inner class

// Action for disconnecting from server
private class DisconnectAction extends AbstractAction {

    // configure DisconnectAction
    public DisconnectAction()
    {
        putValue( Action.NAME, "Disconnect" );
        putValue( Action.SMALL_ICON, new ImageIcon( ClientGUI.class.getResource( "images/Disconnect.gif" ) ) );
    }

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger client (part 5 of 9).
putValue( Action.SHORT_DESCRIPTION,  
"Disconnect from Server" );
putValue( Action.LONG_DESCRIPTION,  
"Disconnect to end Instant Messaging session" );
putValue( Action.MNEMONIC_KEY, new Integer( 'D' ) );
}

// disconnect from server
public void actionPerformed( ActionEvent event )
{
    // disconnect MessageManager from server
    try {

        // send message indicating user disconnected
        messageManager.sendMessage( userName, userName +  
            " exited chat" );

        // disconnect from server and unregister
        // MessageListener
        messageManager.disconnect( messageListener );

        // update Actions, inputArea and statusBar
        sendAction.setEnabled( false );
        disconnectAction.setEnabled( false );
        inputArea.setEditable( false );
        connectAction.setEnabled( true );
        statusBar.setText( "Not Connected" );

    } // end try

    // handle exception disconnecting from server
    catch ( Exception exception ) {
        JOptionPane.showMessageDialog( ClientGUI.this,  
            "Unable to disconnect from server.",  
            "Error Disconnecting", JOptionPane.ERROR_MESSAGE );

        exception.printStackTrace();
    }
}
} // end method actionPerformed
} // end DisconnectAction inner class

// Action for sending messages
private class SendAction extends AbstractAction {

    // configure SendAction
    public SendAction()
    {
        putValue( Action.NAME, "Send" );
        putValue( Action.SMALL_ICON, new ImageIcon(  
            ClientGUI.class.getResource( "images/Send.gif" ) ) );

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger  
client (part 6 of 9).
Chapter 13 Remote Method Invocation 833

```java
269  putValue( Action.SHORT_DESCRIPTION, "Send Message" );
270  putValue( Action.LONG_DESCRIPTION,
271   "Send an Instant Message" );
272  putValue( Action.MNEMONIC_KEY, new Integer( 'S' ) );
273  }
274
275  // send message and clear inputArea
276  public void actionPerformed( ActionEvent event )
277  {
278    // send message to server
279    try {
280      // send userName and text in inputArea
281      messageManager.sendMessage( userName,
282          inputArea.getText() );
283      inputArea.setText( "" );
284    }
285  }
286
287  // handle exception sending message
288  catch ( Exception exception ) {
289    JOptionPane.showMessageDialog( ClientGUI.this,
290      "Unable to send message.", "Error Sending Message",
291      JOptionPane.ERROR_MESSAGE );
292    exception.printStackTrace();
293  }
294
295  } // end method actionPerformed
296
297 } // end SendAction inner class
298
299 // MyMessageListener listens for new messages from the
300 // MessageManager and displays the messages in messageArea
301 // using a MessageDisplayer.
302 private class MyMessageListener implements MessageListener {
303  // when new message received, display in messageArea
304  public void messageReceived( String from, String message )
305  {
306    // append message using MessageDisplayer and invokeLater
307    // to ensure thread-safe access to messageArea
308    SwingUtilities.invokeLater(
309      new MessageDisplayer( from, message ) );
310  }
311 }
312
313 } // end MyMessageListener inner class
314
315 // MessageDisplayer displays a new message by appending
316 // the message to the messageArea JTextArea. This Runnable
317 // object should be executed only on the event-dispatch
318 // thread, as it modifies a live Swing component.
```

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger client (part 7 of 9).
private class MessageDisplayer implements Runnable {
    private String fromUser;
    private String messageBody;

    // MessageDisplayer constructor
    public MessageDisplayer( String from, String body ) {
        fromUser = from;
        messageBody = body;
    }

    // display new message in messageArea
    public void run() {
        // append new message
        messageArea.append( new String( fromUser + " > " + messageBody ) );

        // move caret to end of messageArea to ensure new
        // message is visible on screen
        messageArea.setCaretPosition( messageArea.getText().length() );
    }

} // end MessageDisplayer inner class

// DisconnectHandler listens for serverDisconnected messages
// from the MessageManager and updates the user interface.
private class DisconnectHandler implements DisconnectListener {

    // receive disconnect notification
    public void serverDisconnected( final String message ) {
        // update GUI in thread-safe manner
        SwingUtilities.invokeLater( new Runnable() {

            // update Actions, inputs and status bar
            public void run() {
                sendAction.setEnabled( false );
                disconnectAction.setEnabled( false );
                inputArea.setEditable( false );
                connectAction.setEnabled( true );
                statusBar.setText( message );
            }
        } );
    }

} // end DisconnectHandler inner class

Fig. 13.23 ClientGUI provides a graphical user interface for the Deitel Messenger
client (part 8 of 9).
The ClientGUI constructor (lines 41–141) creates and lays out the various user-interface components. The constructor takes as an argument the MessageManager that implements the underlying network communications. A WindowAdapter inner class (lines 119–138) ensures that the MessageManager disconnects from the ChatServer (line 126) when the user closes the application window.

The ConnectAction inner class (lines 144–205) is an Action implementation for connecting to the Deitel Messenger server. Lines 170–174 invoke method connect of interface MessageManager and register a DisconnectListener for receiving serverDisconnected notifications. Lines 177–186 prompt the user for a name to use in the chat session and update the user-interface components to allow the user to send messages and disconnect from the Deitel Messenger server. Lines 188–189 invoke method sendMessage of interface MessageManager to send a ChatMessage that announces the user’s arrival in the chat session.

The DisconnectAction inner class (lines 211–258) is an Action implementation for disconnecting the MessageManager from the Deitel Messenger server. Lines 231–232 send a ChatMessage to announce the user’s departure from the chat session. Line 236 invokes method disconnect of interface MessageManager to disconnect from the server. Lines 239–243 update the user-interface components to disable the message inputArea and display a message in the status bar.

The SendAction inner class (lines 261–299) is an Action implementation for sending messages to the server. Lines 282–283 invoke method sendMessage of interface MessageManager and pass the contents of inputArea and the user’s userName as arguments.

An instance of inner class MyMessageListener (lines 304–315) listens for incoming ChatMessages. When the MessageManager receives a new ChatMessage from the server, the MessageManager invokes method messageReceived (lines 307–313). Lines 311–312 invoke static method invokeLater of class SwingUtilities with a MessageDisplayer argument to display the new message.

Inner class MessageDisplayer (lines 321–346) is a Runnable implementation that appends a new message to the messageArea JTextArea to display that message to the user. Lines 337–338 append the message text and sender’s user name to messageArea, and lines 342–343 move the cursor to the end of messageArea.

An instance of inner class DisconnectHandler (lines 350–375) receives serverDisconnected notifications from the MessageManager when the server disconnects. Lines 356–371 update the user-interface components to indicate that the server disconnected.

Class DeitelMessenger (Fig. 13.24) launches the client application using a ClientGUI and RMIMessageManager. Line 18 invokes method setSecurityManager of class System to install an RMISecurityManager for the client application.
The client requires this `SecurityManager` for downloading the `ChatServer`’s stub dynamically. We discuss dynamic class downloading in Section 13.6.3. If the user does not specify a hostname for the `ChatServer`, line 24 creates an `RMIMessageManager` that connects to the server running on `localhost`. Line 26 creates an `RMIMessageManager` that connects to the user-provided hostname. Lines 29–32 create a `ClientGUI` for the `RMIMessageManager` and display that GUI to the user.

### 13.6.3 Running the Deitel Messenger Server and Client Applications

Running the Deitel Messenger case study server and clients requires several steps. In addition to the RMI registry, RMI applications that use `Activatable` objects require the RMI activation daemon (`rmid`). The RMI activation daemon is a server process that manages the registration, activation and deactivation of `Activatable` remote objects.

```java
// DeitelMessenger.java
// DeitelMessenger uses a ClientGUI and RMIMessageManager to
// implement an RMI-based chat client.
package com.deitel.messenger.rmi.client;

// Java core packages
import java.rmi.RMISecurityManager;

// Deitel packages
import com.deitel.messenger.*;

public class DeitelMessenger {
    // launch DeitelMessenger application
    public static void main ( String args[] ) throws Exception {
        // install RMISecurityManager
        System.setSecurityManager( new RMISecurityManager() );

        MessageManager messageManager;

        // create new DeitelMessenger
        if ( args.length == 0 )
            messageManager = new RMIMessageManager( "localhost" );
        else
            messageManager = new RMIMessageManager( args[ 0 ] );

        // finish configuring window and display it
        ClientGUI clientGUI = new ClientGUI( messageManager );
        clientGUI.pack();
        clientGUI.setResizable( false );
        clientGUI.setVisible( true );
    }
}
```

*Fig. 13.24* DeitelMessenger launches a chat client using classes `ClientGUI` and `RMIMessageManager`. 
To begin, start the RMI registry by executing the command

```
rmiregistry
```

at a command prompt. Be sure that the stub file for the ChatServer remote object (ChatServerImpl_Stub.class) is not in the RMI registry’s CLASSPATH, as this will disable dynamic class downloading. Next, start the RMI activation daemon by executing the command

```
rmid -J-Djava.security.policy=rmid.policy
```

where rmid is the complete path to the policy file of Fig. 13.25. This policy file allows the ActivationGroup in which the ChatServer runs to specify C:\activationGroup.policy as the policy file for the ActivationGroup’s virtual machine. If you place activationGroup.policy in a location other than the C:\ directory, be sure to modify rmid.policy to specify the appropriate location.

Dynamic class downloading enables Java programs to download classes not available in the local CLASSPATH. This is particularly useful in RMI applications for enabling clients to download stub files dynamically. When an RMI object specifies the java.rmi.server.codebase system property, the RMI registry adds an annotation to that object’s remote references. This annotation specifies the codebase from which clients can download any necessary classes. These classes might include the stub for the remote object and other classes. These .class files must be available for download from an HTTP server. Sun provides a basic HTTP server suitable for testing purposes, which is downloadable from

```
java.sun.com/products/jdk/rmi/class-server.zip
```

Extract the files from class-server.zip and read the included instructions for running the HTTP server. Figure 13.26 lists the files to include in the HTTP server’s download directory. For example, if the HTTP server’s download directory is C:\classes, copy the directory structure and .class files listed in Fig. 13.26 to C:\classes. Be sure to start the HTTP server before continuing.

Next, run the ChatServerAdministrator application to launch the Activatable remote object by using the command

```
java -Djava.security.policy=administrator.policy
-Djava.rmi.server.codebase=http://hostname:port/
com.deitel.messenger.rmi.server.ChatServerAdministrator
start
```

```
1 // allow ActivationGroup to specify C:\activationGroup.policy
2 // as its VM’s security policy
3 grant {
4     permission com.sun.rmi.rmid.ExecOptionPermission
5        "-Djava.security.policy=file:///C:/activationGroup.policy";
6 }
```

Fig. 13.25 Policy file for the RMI activation daemon.
where \texttt{administrator.policy} is the complete path to the policy file of Fig. 13.27, \texttt{hostname} is the name of the computer running the HTTP server and \texttt{port} is the port number on which that HTTP server is running. The RMI registry will annotate each remote reference it returns with this codebase. The policy file must permit \texttt{ChatServerAdministrator} to connect to port 1098 on the local machine, which is the port for the RMI activation daemon. The policy file also must allow the \texttt{ChatServerAdministrator} to access the port on which the Web server is running. Lines 4–5 of Fig. 13.27 specify that the \texttt{ChatServerAdministrator} can access all ports above and including 1024 on \texttt{hostname}. Be sure to replace \texttt{hostname} with the appropriate name or IP address of the machine running the Web server and RMI activation daemon. The \texttt{ChatServerAdministrator} also requires the permission \texttt{setFactory} of type \texttt{java.lang.RuntimePermission}, which permits the \texttt{ActivationGroup} to set a \texttt{SecurityManager}.

The \texttt{ChatServerAdministrator} application registers the \texttt{ActivationGroup} for the \texttt{Activatable ChatServer}, then exits. Clients then may access the \texttt{ChatServer} by obtaining a remote reference to the \texttt{ChatServer} from the RMI registry and invoking methods on that remote reference. Note that the \texttt{ChatServer} does not begin executing until the first client invokes a method on the \texttt{ChatServer} remote object.
At that time, the activation system activates the ChatServer’s ActivationGroup. To launch a client for the ChatServer, type the following at a command prompt:

```
java -Djava.security.policy=client.policy
com.deitel.messenger.rmi.client.DeitelMessenger
```

where `client.policy` is the policy file of Fig. 13.28. This policy file enables the client to connect, accept and resolve connections to the specified hostname on ports above and including 1024. Recall that the client is itself a remote object, so the client must be able to accept incoming network connections from the ChatServer. Be sure to replace `hostname` with the hostname or IP address of the computer on which the ChatServer is running.

Figure 13.29 shows a sample conversation in Deitel Messenger. Notice that the GUI elements properly reflect the current connection state—when the client is disconnected, only the ConnectAction is enabled. After the client connects, the DisconnectAction, input JTextArea and SendAction become enabled. Note also that the bottom of each window displays the message Java Applet Window. The virtual machine places this message in the windows because the application is running under security restrictions.

```java
1 // allow client to connect to network resources on hostname
2 // at ports above 1024
3 grant {
4     permission java.net.SocketPermission "hostname:1024-",
5         "connect, accept, resolve";
6 }
```

Fig. 13.28 Policy file for the DeitelMessenger client.

Fig. 13.29 Sample conversation using Deitel Messenger.
13.7 Internet and World Wide Web Resources

java.sun.com/products/jdk/rmi/index.html
Sun’s Remote Method Invocation (RMI) home page, which provides links to technical articles, documentation and other resources.

java.sun.com/j2se/1.3/docs/guide/rmi/index.html
Sun’s RMI guide, which includes links to tutorials on building activatable remote objects and other useful resources.

www.jguru.com/faq/home.jsp?topic=RMI
jGuru’s RMI Frequently Asked Questions with answers, which provides tips and answers to many common questions that developer’s ask about RMI.

www.javaworld.com/javaworld/topicalindex/jw-ti-rmi.html
JavaWorld’s list of articles related to RMI. Articles include discussions of activatable RMI objects, integrating RMI with CORBA and RMI-related technologies, such as Jini.
SUMMARY

- RMI allows Java objects running on separate computers or in separate processes to communicate with one another via remote method calls. Such method calls appear to the programmer the same as those operating on objects in the same program.

- RMI is based on a similar, earlier technology for procedural programming called remote procedure calls (RPCs) developed in the 1980s.

- RMI enables Java programs to transfer complete Java objects using Java’s object-serialization mechanism. The programmer need not be concerned with the transmission of the data over the network.

- For Java-to-non-Java communication, you can use Java IDL (introduced in Java 1.2) or RMI-IIOP. Java IDL and RMI-IIOP enable applications and applets written in Java to communicate with objects written in any language that supports CORBA (Common Object Request Broker Architecture).

- The four major steps for building an RMI distributed system are 1) defining the remote interface, 2) defining the remote object implementation, 3) defining the client application that uses the remote object and 4) compiling and executing the remote object and the client.

- To create a remote interface, define an interface that extends interface `java.rmi.Remote`. Interface `Remote` is a tagging interface—it does not declare any methods, and therefore places no burden on the implementing class.

- An object of a class that implements interface `Remote` directly or indirectly is a remote object and can be accessed—with appropriate security permissions—from any Java virtual machine that has a connection to the computer on which the remote object executes.

- Every remote method must be declared in an interface that extends `java.rmi.Remote`. A remote object must implement all methods declared in its remote interface.

- An RMI distributed application must export an object of a class that implements the `Remote` interface to make that remote object available to receive remote method calls.

- Each method in a `Remote` interface must have a `throws` clause that indicates that the method can throw `RemoteException`s. A `RemoteException` indicates a problem communicating with the remote object.

- RMI uses Java’s default serialization mechanism to transfer method arguments and return values across the network. Therefore, all method arguments and return values must be `Serializable` or primitive types.

- Class `UnicastRemoteObject` provides the basic functionality required for all remote objects. In particular, its constructor exports the object to make it available to receive remote calls.

- Exporting a remote object enables that object to wait for client connections on an anonymous port number (i.e., one chosen by the computer on which the remote object executes). RMI abstracts away communication details so the programmer can work with simple method calls.

- Constructors for class `UnicastRemoteObject` allow the programmer to specify information about the remote object, such as an explicit port number on which to export the remote object. All `UnicastRemoteObject` constructors throw `RemoteException`s.

- The `rmiregistry` utility program manages the registry for remote objects and is part of the J2SE SDK. The default port number for the RMI registry is 1099.

- Method `lookup` connects to the RMI registry and returns a `Remote` reference to the remote object. Note that clients refer to remote objects only through those object’s remote interfaces.

- A remote reference refers to a stub object on the client. Stubs allow clients to invoke remote objects’ methods. Stub objects receive each remote method call and pass those calls to the RMI system, which performs the networking that allows clients to interact with the remote object.
• The RMI layer is responsible for network connections to the remote object, so referencing remote objects is transparent to the client. RMI handles the underlying communication with the remote object and the transfer of arguments and return values between the objects. Argument and return types for remote methods must be **Serializable**.

• The `rmic` utility compiles the remote object class to produce a stub class. A stub class forwards method invocations to the RMI layer, which performs the network communication necessary to invoke the method call on the remote object.

• Standard RMI objects exported as `UnicastRemoteObjects` must run continuously on the server to handle client requests. RMI objects that extend class `java.rmi.activation.Activatable` are able to activate, or start running, when a client invokes one of the remote object’s methods.

• The RMI activation daemon (`rmid`) is a server process that enables activatable remote objects to become active when clients invoke remote methods on these objects.

• Activatable remote objects also are able to recover from server crashes, because remote references to activatable objects are persistent—when the server restarts, the RMI activation daemon maintains the remote reference, so clients can continue to use the remote object.

• The RMI activation mechanism requires that `Activatable` objects provide a constructor that takes as arguments an `ActivationID` and a `MarshalledObject`. When the activation daemon activates a remote object of this class, it invokes this activation constructor. The `ActivationID` argument specifies a unique identifier for the remote object.

• Class `MarshalledObject` is a wrapper class that contains a serialized object for transmission over RMI. The `MarshalledObject` passed to the activation constructor can contain application-specific initialization information, such as the name under which the activation daemon registered the remote object.

• Activatable RMI objects execute as part of an `ActivationGroup` (package `java.rmi.activation`). The RMI activation daemon—a server-side process that manages activatable objects—starts a new virtual machine for each `ActivationGroup`.

• Class `ActivationGroupDesc` specifies configuration information for an `ActivationGroup`. The first argument to the `ActivationGroupDesc` constructor is a `Properties` reference that contains replacement values for system properties in the `ActivationGroup`’s virtual machine. The second argument is a reference to an `ActivationGroupDesc.CommandEnvironment` object, which enables the `ActivationGroup` to customize the commands that the activation daemon executes when starting the `ActivationGroup`’s virtual machine.

• The incarnation number of an `ActivationGroup` identifies different instances of the same `ActivationGroup`. Each time the activation daemon activates the `ActivationGroup`, the daemon increments the incarnation number.

• Class `ActivationDesc` specifies configuration information for a particular `Activatable` remote object. The first argument to the `ActivationDesc` constructor specifies the name of the class that implements the `Activatable` remote object. The second argument specifies the codebase that contains the remote object’s class files. The final argument is a `MarshalledObject` reference, whose object specifies initialization information for the remote object.

• Method `register` of class `Activatable` takes as an argument the `ActivationDesc` for the `Activatable` object and returns a reference to the remote object’s stub.

• Dynamic class downloading enables Java programs to download classes not available in the local `CLASSPATH`. This is particularly useful in RMI applications for enabling clients to download stub files dynamically.

• When an RMI object specifies the `java.rmi.server.codebase` system property, the RMI registry adds an annotation to that object’s remote references, which specifies the codebase from
which clients can download necessary classes. Downloadable \texttt{.class} files must be available from an HTTP server.

\textbf{TERMINOLOGY}

\begin{itemize}
\item \textbf{Activatable} class (package \texttt{java.rmi.activation})
\item activatable remote object
\item activation daemon
\item activation descriptor
\item activation group descriptor
\item \textbf{ActivationGroup} class
\item \textbf{ActivationGroupDesc} class
\item \textbf{ActivationGroupDescd.CommandEnvironment} class
\item \textbf{ActivationID} class
\item \textbf{ActivationSystem} interface
\item Adapter design pattern
\item anonymous port number
\item \textbf{bind} method of class \textbf{Naming}
\item \textbf{createRegistry} method of class \textbf{LocateRegistry}
\item distributed computing
\item \textbf{export} method
\item \textbf{exportObject} method of class \textbf{UnicastRemoteObject}
\item \textbf{HTML scraping}
\item Interface Definition Language (IDL)
\item \textbf{ListCellRenderer} interface
\item \textbf{LocateRegistry} class
\item \textbf{marshaling} of data
\item \textbf{MarshalledObject} class
\item \textbf{rebind} method of class \textbf{Naming}
\item \textbf{Registry} class
\item remote interface
\item \textbf{Remote} interface (package \texttt{java.rmi})
\item remote method
\item remote method call
\item Remote Method Invocation (RMI)
\item remote object
\item remote object implementation
\item Remote Procedure Call (RPC)
\item remote reference
\item \textbf{RemoteException} class (package \texttt{java.rmi})
\item RMI registry
\item \textbf{rmic} compiler
\item \textbf{rmid} utility
\item \textbf{rmiregistry} utility
\item \textbf{RMISecurityManager} class
\item stub class
\item tagging interface
\item \textbf{UnicastRemoteObject} class (package \texttt{java.rmi.server})
\end{itemize}

\textbf{SELF-REVIEW EXERCISES}

13.1 Fill in the blanks in each of the following statements:

a) The remote object class must be compiled using the \underline{stub class} to produce a stub class.
b) RMI is based on a similar technology for procedural programming called \underline{Remote Method Invocation (RMI)}.
c) Clients use method \underline{bind} of class \textbf{Naming} to obtain a remote reference to a remote object.
d) To create a remote interface, define an interface that extends interface \underline{interface} of package \texttt{java.rmi}.
e) Method \underline{bind} or \underline{rebind} of class \textbf{Naming} binds a remote object to the RMI registry.
f) Remote objects normally extend class \underline{remote object}, which provides the basic functionality required for all remote objects.
g) Remote objects use the \underline{RMI registry} and \underline{remote reference} to locate the RMI registry so they can register themselves as remote services. Clients use these to locate a service.
h) The default port number for the RMI registry is \underline{1099}.
i) Interface \textbf{Remote} is a \underline{interface}.
j) \underline{Remote Method Invocation} allows Java objects running on separate computers (or possibly the same computer) to communicate with one another via remote method calls.
13.2 State whether each of the following is true or false. If false, explain why.
   a) Not starting the RMI registry before attempting to bind the remote object to the registry results in a `RuntimeException` refusing connection to the registry.
   b) Every remote method must be part of an interface that extends `java.rmi.Remote`.
   c) The `stubcompiler` creates a stub class that performs the networking which allows the client to connect to the server and use the remote object’s methods.
   d) Class `UnicastRemoteObject` provides basic functionality required by remote objects.
   e) An object of a class that implements interface `Serializable` can be registered as a remote object and receive a remote method call.
   f) All methods in a `Remote` interface must have a `throws` clause indicating the potential for a `RemoteException`.
   g) RMI clients assume that they should connect to port 80 on a server computer when attempting to locate a remote object through the RMI registry.
   h) Once a remote object is bound to the RMI registry with method `bind` or `rebind` of class `Naming`, the client can look up the remote object with `Naming` method `lookup`.
   i) Method `find` of class `Naming` interacts with the RMI registry to help the client obtain a reference to a remote object so the client can use the remote object’s services.

**ANSWERS TO SELF-REVIEW EXERCISES**

13.1 a) `rmic` compiler.  b) RPC.  c) `lookup`.  d) `Remote`, `java.rmi`.  e) `bind`, `rebind`.  
f) `UnicastRemoteObject`.  g) host, port.  h) 1099.  i) tagging interface.  j) RMI.

13.2 a) False. This results in a `java.rmi.ConnectException`.
   b) True.
   c) False. The `rmic` compiler creates a stub class.
   d) True.
   e) False. An object of a class that implements a subinterface of `java.rmi.Remote` can be registered as a remote object and receive remote method calls.
   f) True.
   g) False. RMI clients assume port 1099 by default. Web browser clients assume port 80.
   h) True.
   i) False. Method `lookup` interacts with the RMI registry to help the client obtain a reference to a remote object.

**EXERCISES**

13.3 The current implementation of class `WeatherServiceImpl` downloads the weather information only once. Modify class `WeatherServiceImpl` to obtain weather information from the National Weather Service twice a day.

13.4 Modify interface `WeatherService` to include support for obtaining the current day’s forecast and the next day’s forecast. Study the Traveler’s Forecast Web page

   `http://iwin.nws.noaa.gov/iwin/us/traveler.html`

13.5 Visit the NWS Web site for the format of each line of information. Next, modify class `WeatherServiceImpl` to implement the new features of the interface. Finally, modify class `WeatherServiceClient` to allow the user to select the weather forecast for either day. Modify the support classes `WeatherBean` and `WeatherItem` as necessary to support the changes to classes `WeatherServiceImpl` and `WeatherServiceClient`. 
13.6 (Project: Weather for Your State) There is a wealth of weather information on the National Weather Service Web site. Study the following Web pages:

http://iwin.nws.noaa.gov/
http://iwin.nws.noaa.gov/iwin/textversion/main.html

and create a complete weather forecast server for your state. Design your classes for reusability.

13.7 (Project: Weather for Your State) Modify the Exercise 13.6 project solution to allow the user to select the weather forecast for any state. [Note: For some states, the format of the weather forecast differs from the standard format. Your solution should allow the user to select only from those states whose forecasts are in the standard format.]

13.8 (For International Readers) If there is a similar World Wide Web-based weather service in your own country, provide a different WeatherServiceImpl implementation with the same remote interface WeatherService (Fig. 13.1). The server should return weather information for major cities in your country.

13.9 (Remote Phone Book Server) Create a remote phone book server that maintains a file of names and phone numbers. Define interface PhoneBookServer with the following methods:

```
public PhoneBookEntry[] getPhoneBook()
public void addEntry( PhoneBookEntry entry )
public void modifyEntry( PhoneBookEntry entry )
public void deleteEntry( PhoneBookEntry entry )
```

Create Activatable remote object class PhoneBookServerImpl, which implements interface PhoneBookServer. Class PhoneBookEntry should contain String instance variables that represent the first name, last name and phone number for one person. The class should also provide appropriate set/get methods and perform validation on the phone number format. Remember that class PhoneBookEntry also must implement Serializable, so that RMI can serialize objects of this class.

13.10 Class PhoneBookClient should provide a user interface that allows the user to scroll through entries, add a new entry, modify an existing entry and delete an existing entry. The client and the server should provide proper error handling (e.g., the client cannot modify an entry that does not exist).
Session EJBs and Distributed Transactions

Objectives

- To understand EJBs as business-logic components.
- To understand the advantages and disadvantages of stateful and stateless session EJBs.
- To understand JNDI’s role in enterprise Java applications.
- To understand distributed transactions.
- To understand the advantages and disadvantages of container-managed and bean-managed transaction demarcation.

*Only the traveling is good which reveals to me the value of home and enables me to enjoy it better.*
Henry David Thoreau

*Youth would be an ideal state if it came a little later in life.*
Herbert Henry Asquith

*We cannot make events. Our business is wisely to improve them.*
Samuel Adams
14.1 Introduction
In previous chapters, we presented Java servlets and JavaServer Pages for implementing business and presentation logic in multi-tier applications. In this chapter, we introduce Enterprise JavaBeans (EJBs), which provide a component model for building business logic in enterprise Java applications.

In this chapter, we introduce session EJBs in their two forms: stateful and stateless. We also introduce EJB support for distributed transactions, which help to ensure data integrity across databases and across application servers. After reading this chapter, you will be able to develop stateless and stateful session EJBs. You also will be able to build EJBs that take advantage of J2EE’s distributed transaction support to update data across multiple databases atomically.

14.2 EJB Overview
Every EJB consists of a remote interface, a home interface and an EJB implementation. The remote interface declares business methods that clients of the EJB may invoke. The home interface provides create methods for creating new EJB instances, finder methods for finding EJB instances and remove methods for removing EJB instances. The EJB implementation defines the business methods declared in the remote interface and the create, remove and finder methods of the home interface. An EJB container provides the EJB’s runtime environment and life-cycle management.
The J2EE specification defines six roles for people implementing enterprise systems. Each role is responsible for producing some part of an enterprise application. The enterprise bean provider implements the Java classes for EJBs. The application assembler constructs application components from EJBs implemented by the enterprise bean provider. The deployer takes application components provided by the application assembler and deploys the application to an EJB container, ensuring that all dependencies are met. The EJB server provider and EJB container provider implement an application server suitable for the deployment of J2EE applications. An application server typically includes an EJB container and a servlet container and provides services such as JNDI directories, database connection pooling, integration with distributed systems and resource management. The same developer or team of developers can play more than one role in the construction and deployment of an enterprise application. For more information on the various roles in building J2EE applications, please see the resources listed in Section 14.5.

14.2.1 Remote Interface

The EJB remote interface declares business methods that EJB clients can invoke. The remote interface must extend interface `javax.ejb.EJBObject`. The EJB container generates a class that implements the remote interface. This generated class implements interface `EJBObject` methods and delegates business method invocations to the EJB implementation (Section 14.2.3). Figure 14.1 describes interface `EJBObject` methods.

Each remote-interface method is required to declare that it throws `RemoteException`. Each method also may throw application-specific exceptions—for example, an `IllegalArgumentException` if a provided argument does not meet certain criteria.

14.2.2 Home Interface

The EJB home interface declares methods for creating, removing and finding EJB instances. The home interface must extend interface `javax.ejb.EJBHome`. The EJB container provides an implementation of the home interface. Depending on the type of EJB (i.e., session or entity), the container will invoke EJB implementation methods that correspond to methods `create` and `remove` and `finder` methods of the home interface. These `finder` methods enable clients to locate a particular instance of an EJB. Figure 14.2 describes interface `EJBHome` methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getEJBHome</td>
<td>Returns the <code>EJBHome</code> interface for the <code>EJBOBJECT</code>.</td>
</tr>
<tr>
<td>getHandle</td>
<td>Returns a <code>Handle</code> for the <code>EJBOBJECT</code>. A <code>Handle</code> is a persistent, <code>Serializable</code> reference to an <code>EJBOBJECT</code>.</td>
</tr>
<tr>
<td>getPrimaryKey</td>
<td>Returns the <code>EJBOBJECT</code>’s primary key if <code>EJBOBJECT</code> is an entity bean.</td>
</tr>
<tr>
<td>isIdentical</td>
<td>Returns a <code>boolean</code> indicating if the <code>EJBOBJECT</code> argument is identical to the current <code>EJBOBJECT</code>.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes the <code>EJBOBJECT</code>.</td>
</tr>
</tbody>
</table>

Fig. 14.1 Methods of interface `javax.ejb.EJBOBJECT`. 
Chapter 14  Session EJBs and Distributed Transactions  849

14.2.3 EJB Implementation

The EJB implementation defines the business methods declared in the EJB remote interface and the create, remove and finder methods declared in the EJB home interface. Session EJBs also must implement interface javax.ejb.SessionBean. We discuss interface SessionBean in detail in Section 14.3.

14.2.4 EJB Container

The EJB container manages an EJB’s client interactions, method invocations, transactions, security, exceptions, etc. Clients of an EJB do not interact directly with the EJB. Clients access the EJB container to obtain remote references to EJB instances. When a client invokes an EJB business method, the invocation goes first to the EJB container, which then delegates the business method invocation to the EJB implementation.

The EJB container also manages the life cycles of its EJBs. EJB containers typically perform pooling of EJB instances to enhance performance. By maintaining a pool of inactive EJB instances, the EJB container can increase performance by avoiding the overhead associated with creating new EJB instances for each client request. The EJB container simply activates a pooled instance and performs any necessary initialization. The EJB container also can create new EJB instances and remove existing instances. In addition, the EJB container provides more advanced services for entity EJBs (Chapter 15, Entity EJBs).

14.3 Session Beans

A session EJB instance performs business logic processing for a particular client. Session EJBs can manipulate data in a database, but unlike entity EJBs (Chapter 15), session EJBs are not persistent and do not represent database data directly. Session EJB instances are lost if the EJB container crashes. There are two session EJB types: stateful and stateless. Section 14.3.1 presents stateful session EJBs and Section 14.3.3 presents stateless session EJBs. Section 14.3.2 discusses deploying session EJBs to the J2EE 1.2.1 reference implementation.

14.3.1 Stateful Session EJBs

Stateful session EJBs maintain state information between business method invocations. For example, a stateful session EJB could maintain information about a customer’s shopping cart while the customer browses an on-line store. The stateful session EJB would provide business methods for adding and removing items from the shopping cart. Each time the customer added an item to the shopping cart, information about the item, such as its price and

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getEJBMetaData</td>
<td>Returns an EJBMetaData object that provides information about the EJB, such as its home interface class and whether it is a session EJB.</td>
</tr>
<tr>
<td>getHomeHandle</td>
<td>Returns a Handle for the EJBHome interface.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes the EJBOBJECT identified by the given Handle or primary key Object.</td>
</tr>
</tbody>
</table>
quantity, would be stored in the stateful session EJB. If the customer left the site or other-
wise terminated the session, the shopping-cart information would be lost.

**InterestCalculator** (Fig. 14.3) is the remote interface for a stateful session EJB that calculates simple interest. Methods **setPrincipal** (lines 15–16), **setInterestRate** (lines 19–20) and **setTerm** (lines 23–24) set the principal, rate and term values needed to calculate simple interest. Method **getBalance** (line 27) calculates the total balance after interest accrues for the given term. Method **getInterestEarned** (line 30) calculates the amount of interest earned.

Clients of the **InterestCalculator** EJB can invoke only those methods declared in the **InterestCalculator** remote interface. The EJB container for the **InterestCalculator** EJB will create a class that implements the **InterestCalculator** remote interface, including methods declared in interface **javax.ejb.EJBObject**. When a client invokes an **InterestCalculator** remote-interface method, the EJB container will invoke the corresponding method in EJB implementation **InterestCalculatorEJB** (Fig. 14.5). When a client invokes a method declared in interface **javax.ejb.EJBObject**, the container will invoke the corresponding method in the class generated by the EJB container.

```java
// InterestCalculator.java
// InterestCalculator is the remote interface for the
// InterestCalculator EJB.
package com.deitel.advjhtp1.ejb.session.stateful.ejb;

// Java core libraries
import java.rmi.RemoteException;

// Java standard extensions
import javax.ejb.EJBObject;

public interface InterestCalculator extends EJBObject {
    // set principal amount
    public void setPrincipal( double amount )
        throws RemoteException;

    // set interest rate
    public void setInterestRate( double rate )
        throws RemoteException;

    // set loan length in years
    public void setTerm( int years )
        throws RemoteException;

    // get loan balance
    public double getBalance() throws RemoteException;

    // get amount of interest earned
    public double getInterestEarned() throws RemoteException;
}
```

**Fig. 14.3** **InterestCalculator** remote interface for calculating simple interest.
InterestCalculatorHome (Fig. 14.4) is the home interface for the InterestCalculator EJB. InterestCalculatorHome provides method create (lines 15–16) for creating instances of the InterestCalculator EJB. When a client invokes InterestCalculatorHome method create, the EJB container invokes method ejbCreate of class InterestCalculatorEJB (Fig. 14.5). The home interface may declare zero or more create methods. For example, we could declare an additional create method that takes a double argument that initializes the principal amount to use in the simple interest calculation.

InterestCalculatorEJB (Fig. 14.5) implements the business methods declared in the InterestCalculator remote interface. On line 12, InterestCalculatorEJB implements interface SessionBean. This indicates that the InterestCalculatorEJB is a session EJB. Lines 17–19 declare variables that maintain the state of the EJB between business method invocations. The state information consists of the principal amount, the interestRate and the term. Method setPrincipal (lines 22–25) sets the principal amount and stores the value in the principal state variable. Method setInterestRate (lines 28–31) sets the interestRate state variable for the calculation. Method setTerm (lines 34–37) sets the term for which interest will accrue. Method getBalance (lines 40–44) uses the formula

\[ a = p \ (1 + r)^n \]

where

- \( p \) is the principal amount
- \( r \) is the annual interest rate (e.g., .05 for 5%)
- \( n \) is the number of years
- \( a \) is the amount on deposit at the end of the \( n \)th year

To calculate the balance (i.e., amount on deposit). Method getInterestEarned (lines 47–50) calculates the amount of interest earned by subtracting the principal amount from the balance—calculated by method getBalance.
// InterestCalculatorEJB.java
// InterestCalculator is a stateful session EJB for calculating
// simple interest.
package com.deitel.advjhtp1.ejb.session.stateful.ejb;

// Java core libraries
import java.util.*;

// Java standard extensions
import javax.ejb.*;

public class InterestCalculatorEJB implements SessionBean {
    private SessionContext sessionContext;

    // state variables
    private double principal;
    private double interestRate;
    private int term;

    // set principal amount
    public void setPrincipal( double amount )
    {
        principal = amount;
    }

    // set interest rate
    public void setInterestRate( double rate )
    {
        interestRate = rate;
    }

    // set loan length in years
    public void setTerm( int years )
    {
        term = years;
    }

    // get loan balance
    public double getBalance()
    {
        // calculate simple interest
        return principal * Math.pow( 1.0 + interestRate, term );
    }

    // get amount of interest earned
    public double getInterestEarned()
    {
        return getBalance() - principal;
    }
}

Fig. 14.5  InterestCalculatorEJB implementation of InterestCalculator remote interface (part 1 of 2).
Method `setSessionContext` (lines 53–56) is a callback method defined in interface `SessionBean`. The EJB container invokes method `setSessionContext` after creating the EJB instance. Interface `SessionContext` extends interface `EJBContext`, which provides methods for obtaining information about the EJB container.

**Common Programming Error 14.1**

*Returning the *this* reference from a method or passing the *this* reference as an argument is not allowed in an EJB. Instead, use `SessionContext` or `EntityContext` method `getEJBObject` to obtain a reference to the current EJBObject.*

When a client invokes a `create` method in the home interface, the EJB container invokes method `ejbCreate` (line 59). The EJB implementation must provide an `ejbCreate` method for each `create` method declared in the home interface. The `ejbCreate` methods must have the same number and types of arguments as their corresponding `create` methods. The `ejbCreate` methods also must return `void`. For example, if the `InterestCalculatorHome` interface declares a `create` method that takes a double argument for the principal amount, the EJB implementation must define an `ejbCreate` method that takes a double argument. The `InterestCalculator` EJB has an empty implementation of method `ejbCreate`, because no initialization is necessary for this EJB.

The EJB container invokes method `ejbRemove` (line 62) in response to an invocation of method `remove` in the home interface. The EJB container also may invoke method `ejbRemove` if the session expires due to lengthy inactivity. Method `ejbRemove` should free resources the EJB has used.

The EJB container invokes method `ejbPassivate` (line 65) when the container determines that the EJB is no longer needed in memory. The algorithm that the EJB container uses to determine when it should passivate an EJB is application-server dependent. Many application servers enforce a *least recently used policy*, which passivates EJBs that
clients have not accessed recently. When the EJB container passivates an EJB, the container serializes the state of the EJB and removes the EJB from memory.

The EJB container invokes method `ejbActivate` (line 68) to restore an EJB instance that the container passivated previously. The EJB container activates an EJB instance if the client associated with the EJB instance invokes a business method of that EJB instance. The EJB container reads the state information that it saved during passivation and restores the EJB instance in memory.

**Software Engineering Observation 14.1**

*Use the `transient` keyword to mark instance variables that the EJB container should not save and restore during passivation and activation.*

**InterestCalculatorClient** (Fig. 14.6) is an application that uses the `InterestCalculator` EJB to calculate simple interest. Line 25 declares an `InterestCalculator` reference, which `InterestCalculatorClient` uses to calculate simple interest. Method `createInterestCalculator` (lines 71–108) creates an `InterestCalculator` EJB instance to use throughout the application. A client application must use a JNDI directory to *look up* the home interface for an EJB. Line 77 creates an `InitialContext`, which is an interface into a JNDI directory. The `InitialContext` represents a *naming context*, which maps names (e.g., "InterestCalculator") to objects, such as EJBs. Lines 80–81 use method `lookup` of class `InitialContext` to retrieve an `Object` remote reference to the `InterestCalculatorHome` interface. The `String` argument passed to method `lookup` is the name to which the EJB is mapped in the JNDI directory.

```java
// InterestCalculatorClient.java
// InterestCalculatorClient is a GUI for interacting with the
// InterestCalculator EJB.
package com.deitel.advjhtp1.ejb.session.stateful.client;

// Java core libraries
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.text.*;
import java.util.*;

// Java standard extensions
import javax.swing.*;
import javax.rmi.*;
import javax.naming.*;
import javax.ejb.*;

// Deitel libraries
import com.deitel.advjhtp1.ejb.session.stateful.ejb.*;

public class InterestCalculatorClient extends JFrame {

Fig. 14.6 InterestCalculatorClient for interacting with
InterestCalculator EJB (part 1 of 7).
```
// InterestCalculator remote reference
private InterestCalculator calculator;

private JTextField principalTextField;
private JTextField rateTextField;
private JTextField termTextField;
private JTextField balanceTextField;
private JTextField interestEarnedTextField;

// InterestCalculatorClient constructor
public InterestCalculatorClient()
{
    super( "Stateful Session EJB Example" );

    // create InterestCalculator for calculating interest
    createInterestCalculator();

    // create JTextField for entering principal amount
    createPrincipalTextField();

    // create JTextField for entering interest rate
    createRateTextField();

    // create JTextField for entering loan term
    createTermTextField();

    // create uneditable JTextFields for displaying balance
    // and interest earned
    balanceTextField = new JTextField( 10 );
    balanceTextField.setEditable( false );

    interestEarnedTextField = new JTextField( 10 );
    interestEarnedTextField.setEditable( false );

    // layout components for GUI
    layoutGUI();

    // add WindowListener to remove EJB instances when user
    // closes window
    addWindowListener( getWindowListener() );

    setSize( 425, 200 );
    setVisible( true );
}

// create InterestCalculator EJB instance
public void createInterestCalculator()
{
    // lookup InterestCalculatorHome and create
    // InterestCalculator EJB
    try {

Fig. 14.6 InterestCalculatorClient for interacting with InterestCalculator EJB (part 2 of 7).
InitialContext initialContext = new InitialContext();

// lookup InterestCalculator EJB
Object homeObject =
    initialContext.lookup( "InterestCalculator" );

// get InterestCalculatorHome interface
InterestCalculatorHome calculatorHome =
    ( InterestCalculatorHome )
    PortableRemoteObject.narrow( homeObject, InterestCalculatorHome.class );

// create InterestCalculator EJB instance
calculator = calculatorHome.create();

} // end try

// handle exception if InterestCalculator EJB not found
catch ( NamingException namingException ) {
    namingException.printStackTrace();
}

// handle exception when creating InterestCalculator EJB
catch ( RemoteException remoteException ) {
    remoteException.printStackTrace();
}

// handle exception when creating InterestCalculator EJB
catch ( CreateException createException ) {
    createException.printStackTrace();
}
} // end method createInterestCalculator

// create JTextField for entering principal amount
public void createPrincipalTextField()
{
    principalTextField = new JTextField( 10 );

    principalTextField.addActionListener(
        new ActionListener() {

            public void actionPerformed( ActionEvent event )
            {
                // set principal amount in InterestCalculator
                try {
                    double principal = Double.parseDouble(
                        principalTextField.getText() );
                    calculator.setPrincipal( principal );
                }
            }
        }
    );
}

Fig. 14.6 InterestCalculatorClient for interacting with InterestCalculator EJB (part 3 of 7).
// handle exception setting principal amount
catch ( RemoteException remoteException ) {
    remoteException.printStackTrace();
}

// handle wrong number format
catch ( NumberFormatException numberFormatException ) {
    numberFormatException.printStackTrace();
}
}

// create JTextField for entering interest rate
public void createRateTextField() {
    rateTextField = new JTextField( 10 );
    rateTextField.addActionListener(
        new ActionListener() {

            public void actionPerformed( ActionEvent event ) {
                // set interest rate in InterestCalculator
                try {
                    double rate = Double.parseDouble(
                        rateTextField.getText() );

                    // convert from percentage
                    calculator.setInterestRate( rate / 100.0 );
                }
                // handle exception when setting interest rate
                catch ( RemoteException remoteException ) {
                    remoteException.printStackTrace();
                }
            }
        } // end addActionListener
    ); // end addEventListener
}

// create JTextField for entering loan term
public void createTermTextField() {
    termTextField = new JTextField( 10 );
    termTextField.addActionListener(
        new ActionListener() {

            public void actionPerformed( ActionEvent event ) {

            }
        } // end addActionListener
    ); // end addEventListener
}

Fig. 14.6 InterestCalculatorClient for interacting with InterestCalculator EJB (part 4 of 7).
public void actionPerformed( ActionEvent event )
{
    // set loan term in InterestCalculator
    try {
        int term = Integer.parseInt(
            termTextField.getText() );
        calculator.setTerm( term );
    }

    // handle exception when setting loan term
    catch ( RemoteException remoteException ) {
        remoteException.printStackTrace();
    }
}

// get JButton for starting calculation
public JButton getCalculateButton()
{
    JButton calculateButton = new JButton( "Calculate" );
    calculateButton.addActionListener(
        new ActionListener() {
            public void actionPerformed( ActionEvent event )
            {
                // use InterestCalculator to calculate interest
                try {
                    // get balance and interest earned
                    double balance = calculator.getBalance();
                    double interest =
                        calculator.getInterestEarned();

                    NumberFormat dollarFormatter =
                        NumberFormat.getCurrencyInstance(
                            Locale.US );

                    balanceTextField.setText(
                        dollarFormatter.format( balance ) );
                    interestEarnedTextField.setText(
                        dollarFormatter.format( interest ) );
                }

                // handle exception when calculating interest
                catch ( RemoteException remoteException ) {
                    remoteException.printStackTrace();
                }
            }
        });
}

Fig. 14.6 InterestCalculatorClient for interacting with InterestCalculator EJB (part 5 of 7).
231  } // end method actionPerformed
232  
233 }); // end addActionListener
234  
235  return calculateButton;
236  
237 } // end method getCalculateButton
238  
239 // lay out GUI components in JFrame
240  public void layoutGUI()
241  {
242    Container contentPane = getContentPane();
243    
244    // layout user interface components
245    JPanel inputPanel = new JPanel( new GridLayout( 5, 2 ) );
246    
247    inputPanel.add( new JLabel( "Principal" ) );
248    inputPanel.add( principalTextField );
249    
250    inputPanel.add( new JLabel( "Interest Rate (%)" ) );
251    inputPanel.add( rateTextField );
252    
253    inputPanel.add( new JLabel( "Term (years)" ) );
254    inputPanel.add( termTextField );
255    
256    inputPanel.add( new JLabel( "Balance" ) );
257    inputPanel.add( balanceTextField );
258    
259    inputPanel.add( new JLabel( "Interest Earned" ) );
260    inputPanel.add( interestEarnedTextField );
261    
262    // add inputPanel to contentPane
263    contentPane.add( inputPanel, BorderLayout.CENTER );
264    
265    // create JPanel for calculateButton
266    JPanel controlPanel = new JPanel( new FlowLayout() );
267    controlPanel.add( getCalculateButton() );
268    contentPane.add( controlPanel, BorderLayout.SOUTH );
269  }
270  
271 // get WindowListener for exiting application
272  public WindowListener getWindowListener()
273  {
274    return new WindowAdapter() {
275      
276      public void windowClosing( WindowEvent event )
277      {
278        // check to see if calculator is null
279        if ( calculator.equals( null ) ) {
280          System.exit( -1 );
281        }
282    
Fig. 14.6  InterestCalculatorClient for interacting with InterestCalculator EJB (part 6 of 7).
Lines 84–87 use class `PortableRemoteObject` method `narrow` to convert the remote reference to an `InterestCalculatorHome` remote reference. This is the standard method for casting a remote reference to the proper interface type when using RMI-IIOP. RMI-IIOP allows RMI objects to interact with CORBA components, which communicate using the *Internet Inter-Orb Protocol* (IIOP). CORBA is a language-independent framework for building distributed systems. To enable interoperability among EJBs and CORBA components, EJBs communicate using RMI-IIOP. We discuss CORBA and RMI-IIOP in Chapters 26 and 27.
Line 90 invokes method `create` of interface `InterestCalculatorHome` to create a new instance of the `InterestCalculator` EJB. This `InterestCalculator` EJB instance is exclusive to the client that creates it, because it is a session EJB. The `InterestCalculator` EJB instance will maintain state information, because it is a stateful session EJB. Method `create` returns a remote reference to the newly created `InterestCalculator` EJB instance. Line 90 assigns this remote reference to the `calculator` member variable. Instances of class `InterestCalculatorClient` can use this remote reference to invoke the business methods defined in the `InterestCalculator` remote interface.

Lines 95–97 catch a `NamingException`, which indicates a problem accessing the JNDI directory. If the `InitialContext` cannot be created, the `InitialContext` constructor throws a `NamingException`. Method `lookup` also throws a `NamingException` if the name passed as an argument cannot be found in the JNDI directory.

Lines 100–102 catch a `RemoteException`. If there is an error communicating with the EJB container, method `create` throws a `RemoteException`. Lines 105–107 catch a `CreateException`. Method `create` throws a `CreateException` if there is an error creating the EJB instance.

Method `createPrincipalTextField` (lines 111–141) creates a `JTextField` for the user to input the `principal` amount. An `ActionListener` anonymous inner class (lines 116–139) uses `static` method `parseDouble` of class `Double` to get the `principal` entered by the user (lines 122–123). Line 125 sets the `principal` in the `InterestCalculator` EJB by invoking method `setPrincipal`. Note that `JTextField` s generate `ActionEvents` when the user presses the `Enter` key in the `JTextField`. Therefore, the user must press the `Enter` key after entering each piece of data to cause the `ActionListener`'s `actionPerformed` method to invoke the appropriate `InterestCalculator` business method.

Method `createRateTextField` (lines 144–169) creates a `JTextField` for inputting an interest rate for the calculation. Lines 149–167 create an `ActionListener` anonymous inner class to parse the input from the user (lines 155–156) and invoke `InterestCalculator` method `setInterestRate` to set the interest rate (line 159).

Method `createTermTextfield` (lines 172–196) creates a `JTextField` for inputting the number of years for which interest will accrue. An `ActionListener` anonymous inner class (lines 177–194) parses the input from the user (lines 183–184) and sets the term in the `InterestCalculator` by invoking method `setTerm` (line 186) of the `InterestCalculator` remote interface.

Method `getCalculateButton` (lines 199–237) creates a `JButton` that, when clicked, invokes `InterestCalculator` methods `getBalance` and `getInterestEarned`. Lines 220–224 set the text in `balanceTextField` and `interestEarnedTextField` to display the calculation results to the user.

The `InterestCalculatorClient` constructor (lines 34–68) invokes method `createInterestCalculator` to create a new `InterestCalculator` EJB instance (line 39). Lines 42–56 create the application’s user-interface components and invoke method `layoutGUI` (line 59), which creates a `JLabel` for each `JTextField` and lays out the components (lines 240–269).

Method `getWindowListener` (lines 272–305) creates a `WindowAdapter` anonymous inner class that provides method `windowClosing` to perform cleanup tasks when
the user closes the application window. Line 279 checks to see if `calculator` is `null`. If calculator is `null`, the program exits with an error code (line 280). Otherwise, line 286 invokes method `remove` of the `InterestCalculator` remote interface to remove the EJB instance that the application used. If there is an error when removing the EJB instance, lines 290–293 `catch` a `RemoveException`. If there is an error communicating with the EJB container, lines 296–299 `catch` a `RemoteException`.

### 14.3.2 Deploying Session EJBs

Enterprise JavaBeans execute in the context of an EJB container, which is a fundamental part of a J2EE-compliant application server. This section details the steps necessary to deploy the `InterestCalculator` session EJB in the Java 2 Enterprise Edition version 1.2.1 reference implementation from Sun Microsystems. If you have not yet installed and configured the J2EE SDK, please see Appendix E for instructions.

Create a new enterprise application by selecting **New Application** from the **File** menu (Fig. 14.7). Specify the file name for the EAR file and the name of the application in the **New Application** dialog box and click **OK** (Fig. 14.8).

![Fig. 14.7 Creating New Application in Application Deployment Tool.](image)
Select **New Enterprise Bean** from the **File** menu to begin deploying the EJB (Fig. 14.9). Click **Next**, provide a **JAR Display Name** and click **Add** to add the class files for the **InterestCalculator** EJB (Fig. 14.10).

Specify the **Root Directory** for the classes that make up the **InterestCalculator** EJB (Fig. 14.11). The **InterestCalculator** classes are in package `com.deitel.advjhtp1.ejb.session.stateful.ejb`, so select as the **Root Directory** the directory that contains the `com` directory. Select the **InterestCalculator** class files—`InterestCalculator.class`, `InterestCalculator-home.class` and `InterestCalculatorEJB.class`—and click **Add** and **OK**. After adding the class files, they will appear in the **Contents** of the **EJB JAR** (Fig. 14.12). Click **Next** to begin adding class files to the **EJB JAR**.
Fig. 14.10 Adding **InterestCalculator** EJB classes.

Fig. 14.11 Selecting **InterestCalculator** EJB classes to add.
Specify the **Enterprise Bean Class**, **Home Interface** and **Remote Interface** by selecting the appropriate classes from the drop-down lists (Fig. 14.13). Specify an **Enterprise Bean Display Name** to be displayed in the **Application Deployment Tool** and select the **Bean Type**. The **InterestCalculator** EJB is a stateful session bean, so select the **Session** and **Stateful** radio buttons and click **Next** (Fig. 14.14). When deploying the **MathToolEJB**, specify **Stateless** instead of **Stateful** for the **Bean Type**.
Specify **Container-Managed Transactions** in the **Transaction Management** dialog (Fig. 14.15). For each **Method**, specify the **Supports** Transaction Type, and click **Next**. We discuss the details of transactions in Section 14.4.
Figure 14.16 shows the XML deployment descriptor generated for the **InterestCalculator** EJB. The application server uses this XML descriptor to configure the **InterestCalculator** EJB.

You must specify a **JNDI Name** for the **InterestCalculator**, so that clients may obtain references to the EJB. For this example, specify the **JNDI Name InterestCalculator** (Fig. 14.17). The **JNDI Name** need not be the same as the EJB name.

**Good Programming Practice 14.1**

*Use an EJB’s remote interface name as the EJB’s JNDI name. This makes your code more readable and makes the JNDI name easier to remember.*

---

**Fig. 14.16** XML deployment descriptor for **InterestCalculator** EJB.

**Fig. 14.17** Specifying **JNDI Name** for **InterestCalculator** EJB.
Deploy the application to the J2EE server by selecting **Deploy Application** from the **Tools** menu, or by clicking the **Deploy Application** button on the toolbar (Fig. 14.18).

Specify **localhost** as the **Target Server** and select the checkbox **Return Client JAR** (Fig. 14.19). The client JAR contains the stub classes the client will use to communicate with the EJB.

---

**Fig. 14.18** Deploying enterprise application to **localhost**.

**Fig. 14.19** Specifying the **Application Deployment Tool** should **Return Client Jar**.
After deploying the InterestCalculator EJB (Fig. 14.20), execute the InterestCalculatorClient to test the InterestCalculator. The InterestCalculatorClient.jar and the j2ee.jar must be in the CLASSPATH when executing the InterestCalculatorClient. For example, from the command line, type

```
java -classpath D:\j2sdkee1.2.1\lib\j2ee.jar;D:\InterestCalculatorClient.jar;. com.deitel.advjhtp1.ejb.session.stateful.client.InterestCalculatorClient
```

### 14.3.3 Stateless Session EJBs

Stateless session EJBs maintain no state information between business method invocations. As a result, the EJB container can use any stateless session EJB instance to respond to any client’s request.

#### Performance Tip 14.1

Stateless session EJBs may perform better than stateful session EJBs, because a single stateless session EJB instance can be shared among many clients, reducing memory, processor and other resource requirements on the server.

Figure 14.21 shows the MathTool remote interface. MathTool is a stateless session EJB with business methods for generating a Fibonacci series and calculating factorials. Method `getFibonacciSeries` (lines 14–15) generates a Fibonacci series of the length supplied in integer argument `howMany`. Method `getFactorial` (lines 18–19) calculates the factorial of the provided integer.

---

**Fig. 14.20** Successful completion of deployment process.
MathToolEJB (Fig. 14.22) provides implementations of the business methods declared in the MathTool remote interface. MathToolEJB implements the SessionBean interface (line 9), indicating MathTool is a session EJB. Method getFibonacciSeries (lines 14–48) generates a Fibonacci series. The Fibonacci series

\[0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\]

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. The MathTool EJB calculates the Fibonacci series on lines 25–44. Each number in the series is placed in the integer array series (line 29). Lines 32–33 set the zeroth number in the series to 0 and the first number in the series to 1. Line 39 sets the next number in the series to the sum of the previous two numbers.
// get Fibonacci series
public int[] getFibonacciSeries(int howMany) throws IllegalArgumentException {
  // throw IllegalArgumentException if series length is less than zero
  if (howMany < 2)
    throw new IllegalArgumentException("Cannot generate Fibonacci series of " + "length less than two.");

  // starting points
  int startPoint1 = 0;
  int startPoint2 = 1;

  // array to contain Fibonacci sequence
  int[] series = new int[howMany];

  // set base cases
  series[0] = 0;
  series[1] = 1;

  // generate Fibonacci series
  for (int i = 2; i < howMany; i++) {
    // calculate next number in series
    series[i] = startPoint1 + startPoint2;

    // set start points for next iteration
    startPoint1 = startPoint2;
    startPoint2 = series[i];
  }

  return series;
}

// get factorial of given integer
public int getFactorial(int number) throws IllegalArgumentException {
  // throw IllegalArgumentException if number less than zero
  if (number < 0)
    throw new IllegalArgumentException("Cannot calculate factorial of negative numbers.");

  // base case for recursion, return 1
  if (number == 0)
    return 1;

  // end method getFibonacciSeries

  // get factorial of given integer
  public int getFactorial(int number) throws IllegalArgumentException {
    // throw IllegalArgumentException if number less than zero
    if (number < 0)
      throw new IllegalArgumentException("Cannot calculate factorial of negative numbers.");
  
    // base case for recursion, return 1
    if (number == 0)
      return 1;
   
  // end method getFactorial

Fig. 14.22 MathToolEJB implementation of MathTool remote interface (part 2 of 3).
Method `getFactorial` (lines 51–67) calculates the factorial of a nonnegative integer, which is defined to be the product
\[ n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 \]

Lines 56–57 throw an `IllegalArgumentException` if the integer argument `number` is less than 0. Lines 60–61 implement the base case for the recursive calculation. Line 65 calculates the factorial with a recursive call to `getFactorial`.

The EJB container invokes method `setSessionContext` (lines 70–73) when the EJB instance is activated and a remote reference is given to a client. The `SessionContext` argument implements method `getEJBObject`, which the EJB can use to retrieve a reference to the current EJB instance.

When a client invokes method `create` of interface `MathToolHome` (Fig. 14.23), the EJB container invokes method `ejbCreate` (line 76). Method `ejbCreate` performs initialization of the EJB. `MathToolEJB` method `ejbCreate` has an empty implementation, because this EJB requires no initialization. The EJB container invokes method `ejbRemove` (line 79) to remove a `MathTool` EJB instance.

The EJB container invokes method `ejbActivate` (line 82) when the EJB instance is taken out of the EJB container’s pool and associated with a particular client. The EJB container invokes method `ejbPassivate` (line 85) when the EJB instance is no longer needed and can be returned to the ready pool.

`MathToolHome` (Fig. 14.23) is the home interface for the `MathTool` EJB. Method `create` (lines 15–16) creates a new `MathTool` EJB. The EJB container invokes `MathToolEJB` method `ejbCreate` when a client invokes `MathToolHome` method `create`. 

```java
63 // call getFactorial recursively to calculate factorial
64 else
65     return number * getFactorial( number - 1 );
66 }
67 // end method getFactorial
68
69 // set SessionContext
70 public void setSessionContext( SessionContext context )
71 {
72     sessionContext = context;
73 }
74
75 // create new MathTool instance
76 public void ejbCreate() {}
77
78 // remove MathTool instance
79 public void ejbRemove() {}
80
81 // activate MathTool instance
82 public void ejbActivate() {}
83
84 // passivate MathTool instance
85 public void ejbPassivate() {}
86 }
```

Fig. 14.22 `MathToolEJB` implementation of `MathTool` remote interface (part 3 of 3).
Please refer to Section 14.3.2 for deployment instructions. Recall that MathTool is a stateless session EJB, so you must specify this in the deployment process. Also, be sure to specify an appropriate JNDI name (e.g., MathTool) for the MathTool EJB.

MathToolClient (Fig. 14.24) is an application client for the MathTool EJB. The user interface consists of a JTextField into which the user can enter an integer, a JButton to invoke MathTool method getFactorial and a JButton to invoke MathTool method getFibonacciSeries. The calculation results are displayed in a JTextArea.

The MathToolClient constructor (lines 29–45) invokes method createMathTool (line 35) to create a new instance of the MathTool EJB. Line 38 invokes method createGUI to create and lay out GUI components for the application’s user interface. Method createMathTool (lines 48–83) uses an InitialContext (line 53) to look up the MathToolHome interface in the JNDI Directory (lines 56–57). Line 65 invokes MathToolHome method create to create a MathTool EJB instance.

Method getFactorialButton (lines 86–122) creates a JButton that, when pressed, invokes MathTool EJB method getFactorial. Lines 100–104 parse the number entered in numberTextField and invoke method getFactorial of the MathTool remote interface. Lines 107–108 display the factorial in resultsTextArea. Lines 113–115 catch a RemoteException if there is an error invoking method getFactorial.

Method getFibonacciButton (lines 125–182) creates a JButton that, when clicked, invokes MathTool EJB method getFibonacciSeries. Lines 140–145 parse the number entered in numberTextField and invoke method getFibonacciSeries of the MathTool remote interface. Method getFibonacciSeries returns an array of integers containing a Fibonacci series of the length specified by the integer argument howMany. Lines 148–168 build a StringBuffer containing the Fibonacci series and display the series in resultsTextArea.

```
1 // MathToolHome.java
2 // MathToolHome is the home interface for the MathTool EJB.
3 package com.deitel.advjhtp1.ejb.session.stateless.ejb;
4
5 // Java core libraries
6 import java.rmi.RemoteException;
7
8 // Java standard extensions
9 import javax.ejb.EJBHome;
10 import javax.ejb.CreateException;
11
12 public interface MathToolHome extends EJBHome {
13
14    // create new MathTool EJB
15    public MathTool create() throws RemoteException,
16    CreateException;
17 }

Fig. 14.23 MathToolHome interface for creating MathTool EJBs.
```
// MathToolClient.java
// MathToolClient is a GUI for calculating factorials and
// Fibonacci series using the MathTool EJB.
package com.deitel.advjhtp1.ejb.session.stateless.client;

// Java core libraries
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;

// Java standard extensions
import javax.swing.*;
import javax.rmi.*;
import javax.naming.*;
import javax.ejb.*;

// Deitel libraries
import com.deitel.advjhtp1.ejb.session.stateless.ejb.*;

public class MathToolClient extends JFrame {

    private MathToolHome mathToolHome;
    private MathTool mathTool;

    private JTextArea resultsTextArea;
    private JTextField numberTextField;

    // MathToolClient constructor
    public MathToolClient() {
        super( "Stateless Session EJB Example" );

        // create MathTool for calculating factorials
        // and Fibonacci series
        createMathTool();

        // create and lay out GUI components
        createGUI();

        addWindowListener( getWindowListener() );

        setSize( 425, 200 );
        setVisible( true );
    }

    // create MathTool EJB instance
    private void createMathTool() {
        // lookup MathToolHome and create MathTool EJB
        try {
            InitialContext initialContext = new InitialContext();
            ...
        }
    }
}

Fig. 14.24 MathToolClient for interacting with MathTool EJB (part 1 of 6).
// lookup MathTool EJB
Object homeObject = initialContext.lookup( "MathTool" );

// get MathToolHome interface
mathToolHome = ( MathToolHome ) PortableRemoteObject.narrow( homeObject, MathToolHome.class );

// create MathTool EJB instance
mathTool = mathToolHome.create();

} // end try

// handle exception if MathTool EJB is not found
catch ( NamingException namingException ) {
    namingException.printStackTrace();
}

// handle exception when creating MathTool EJB
catch ( RemoteException remoteException ) {
    remoteException.printStackTrace();
}

// handle exception when creating MathTool EJB
catch ( CreateException createException ) {
    createException.printStackTrace();
}

} // end method createMathTool

// create JButton for calculating factorial
private JButton getFactorialButton()
{
    JButton factorialButton =
    new JButton( "Calculate Factorial" );

    // add ActionListener for factorial button
    factorialButton.addActionListener( new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // use MathTool EJB to calculate factorial
            try {

                int number = Integer.parseInt(
                    numberTextField.getText() );

                // get Factorial of number input by user
                int result = mathTool.getFactorial( number );

Fig. 14.24 MathToolClient for interacting with MathTool EJB (part 2 of 6).
// display results in resultsTextArea
resultsTextArea.setText( number + "! = " + result );

} // end try

// handle exception calculating factorial
catch ( RemoteException remoteException ) {
    remoteException.printStackTrace();
}

} // end method actionPerformed

return factorialButton;

} // end method getFactorialButton

// create JButton for generating Fibonacci series
private JButton getFibonacciButton() {
    JButton fibonacciButton =
        new JButton( "Fibonacci Series" );

    // add ActionListener for generating Fibonacci series
    fibonacciButton.addActionListener( new ActionListener() {

        public void actionPerformed( ActionEvent event ) {
            // generate Fibonacci series using MathTool EJB
            try {

                // get number entered by user
                int number = Integer.parseInt( numberTextField.getText() );

                // get Fibonacci series
                int[] series = mathTool.getFibonacciSeries( number );

                // create StringBuffer to store series
                StringBuffer buffer =
                    new StringBuffer( "The first " );

                buffer.append( number );

                buffer.append( " Fibonacci number(s): \
                for ( int i = 0; i < series.length; i++ ) {

Fig. 14.24 MathToolClient for interacting with MathTool EJB (part 3 of 6).
// do not add comma before first number
if ( i != 0 )
    buffer.append( " , ");

// append next number in series to buffer
buffer.append( String.valueOf( series[ i ] ));
}

// display series in resultsTextArea
resultsTextArea.setText( buffer.toString() );

} // end try

// handle exception calculating series
catch ( RemoteException remoteException ) {
    remoteException.printStackTrace();
}

} // end method actionPerformed

return fibonacciButton;

} // end method getFibonacciButton

// create lay out GUI components
public void createGUI()
{

// create JTextArea to show results
resultsTextArea = new JTextArea();
resultsTextArea.setLineWrap( true );
resultsTextArea.setWrapStyleWord( true );
resultsTextArea.setEditable( false );

// create JTextField for user input
numberTextField = new JTextField( 10 );

// create JButton for calculating factorial
JButton factorialButton = getFactorialButton();

// create JButton for generating Fibonacci series
JButton fibonacciButton = getFibonacciButton();

Container contentPane = getContentPane();

// put resultsTextArea in a JScrollPane
JScrollPane resultsScrollPane = new JScrollPane( resultsTextArea );

contentPane.add( resultsScrollPane, BorderLayout.CENTER );

Fig. 14.24 MathToolClient for interacting with MathTool EJB (part 4 of 6).
// add input components to new JPanel
JPanel inputPanel = new JPanel( new FlowLayout() );
inputPanel.add( new JLabel( "Enter an integer: " ) );
inputPanel.add( numberTextField );

// add JButton components to new JPanel
JPanel buttonPanel = new JPanel( new FlowLayout() );
buttonPanel.add( factorialButton );
buttonPanel.add( fibonacciButton );

// add inputPanel and buttonPanel to new JPanel
JPanel controlPanel =
    new JPanel( new GridLayout( 2, 2 ) );
controlPanel.add( inputPanel );
controlPanel.add( buttonPanel );
contentPane.add( controlPanel, BorderLayout.NORTH );

} // end method createGUI

// get WindowListener for exiting application
private WindowListener getWindowListener()
{
    return new WindowAdapter() {

        public void windowClosing( WindowEvent event )
        {
            // remove MathTool instance
            try {
                mathTool.remove();
            }

            // handle exception when removing MathTool EJB
            catch ( RemoveException removeException ) {
                removeException.printStackTrace();
                System.exit( -1 );
            }

            // handle exception when removing MathTool EJB
            catch ( RemoteException remoteException ) {
                remoteException.printStackTrace();
                System.exit( -1 );
            }

        System.exit( 0 );

            } // end method windowClosing
        }
    } // end method getWindowListener

} // execute application
public static void main( String[] args )
{
14.4 EJB Transactions

The Java 2 Enterprise Edition supports distributed transactions. A distributed transaction is a transaction that includes multiple databases or multiple application servers. For example, a distributed transaction could transfer funds from an account at one bank into an account at another bank atomically.

J2EE supports two methods for defining transaction boundaries: bean-managed transaction demarcation and container-managed transaction demarcation. Bean-managed transaction demarcation requires the EJB developer to code the transaction boundaries manually in the EJBs using the Java Transaction API (JTA). Container-managed transaction demarcation allows the EJB deployer to specify transaction boundaries declaratively when deploying EJBs.

Software Engineering Observation 14.2

Entity EJBs may use only container-managed transaction demarcation.

14.4.1 MoneyTransfer EJB Home and Remote Interfaces

The MoneyTransfer EJB demonstrates the need for distributed transactions and their implementation using bean-managed and container-managed transaction demarcation. In this example, we transfer money from an account at BankABC to an account at BankXYZ. We first withdraw money from an account at BankABC and then deposit the same amount at BankXYZ. Transactions are needed to ensure that the money is “put back” in the Bank-
ABC account if the deposit at BankXYZ fails. We also need to ensure that if the withdrawal from BankABC fails, the money is not deposited at BankXYZ.

The MoneyTransfer remote interface (Fig. 14.25) provides methods for transferring money between accounts and for getting the balances of accounts at two different banks. Method transfer (line 15) transfers the given amount of money from an account at BankABC to an account at BankXYZ. Method getBankABCBalance (line 18) returns the account balance at BankABC. Method getBankXYZBalance (line 21) returns the account balance at BankXYZ. Interface MoneyTransferHome (Fig. 14.26) provides method create (lines 15–16) for creating MoneyTransfer EJB instances.

Fig. 14.25 MoneyTransfer remote interface for transferring money and getting account balances.

```java
// MoneyTransfer.java
// MoneyTransfer is the remote interface for the MoneyTransfer EJB.
package com.deitel.advjhtp1.ejb.transactions;

// Java core libraries
import java.rmi.RemoteException;

// Java standard extensions
import javax.ejb.EJBObject;

public interface MoneyTransfer extends EJBObject {

    // transfer amount from BankABC to BankXYZ
    public void transfer( double amount ) throws RemoteException;

    // get BankABC account balance
    public double getBankABCBalance() throws RemoteException;

    // get BankXYZ account balance
    public double getBankXYZBalance() throws RemoteException;
}
```

Fig. 14.26 MoneyTransferHome interface for creating MoneyTransfer EJBs (part 1 of 2).
14.4.2 Bean-Managed Transaction Demarcation

Bean-managed transaction demarcation requires the EJB developer to code the transaction boundaries manually in the EJBs. Bean-managed transaction demarcation may be used only with session EJBs.

MoneyTransferEJB (Fig. 14.27) implements the MoneyTransfer remote interface using bean-managed transaction demarcation to ensure atomicity of the database updates in method transfer (lines 26–81). Lines 29–30 create a UserTransaction. Line 34 begins the transaction by invoking UserTransaction method begin. All statements after the transaction begins are part of the transaction until the transaction is committed or rolled back.

```java
// MoneyTransferEJB.java
// MoneyTransferEJB is a stateless session EJB for transferring funds from an Account at BankABC to an Account at BankXYZ
// using bean-managed transaction demarcation.
package com.deitel.advjhtp1.ejb.transactions.beanmanaged;

// Java core libraries
import java.util.*;
import java.sql.*;

// Java standard extensions
import javax.ejb.*;
import javax.naming.*;
import javax.transaction.*;
import javax.sql.*;

public class MoneyTransferEJB implements SessionBean {
    private SessionContext sessionContext;
    private Connection bankOneConnection;
    private Connection bankTwoConnection;
    private PreparedStatement withdrawalStatement;
    private PreparedStatement depositStatement;

    // transfer funds from BankABC to BankXYZ
    public void transfer( double amount ) throws EJBException
    {
        // create transaction for transferring funds
        UserTransaction transaction =
            sessionContext.getUserTransaction();
```

Fig. 14.26 MoneyTransferHome interface for creating MoneyTransfer EJBs (part 2 of 2).

Fig. 14.27 MoneyTransferEJB implementation of MoneyTransfer remote interface using bean-managed transaction demarcation (part 1 of 6).
// begin bean-managed transaction demarcation
try {
    transaction.begin();
}

// catch exception if method begin fails
catch (Exception exception) {
    // throw EJBException indicating transaction failed
    throw new EJBException(exception);
}

// transfer funds from account in BankABC to account
// in BankXYZ using bean-managed transaction demarcation
try {
    withdrawalStatement.setDouble(1, amount);
    // withdraw funds from account at BankABC
    withdrawalStatement.executeUpdate();
    depositStatement.setDouble(1, amount);
    // deposit funds in account at BankXYZ
    depositStatement.executeUpdate();
    // commit transaction
    transaction.commit();
} // end try

// handle exceptions when withdrawing, depositing and
// committing transaction
catch (Exception exception) {
    // attempt rollback of transaction
    try {
        transaction.rollback();
    }
    // handle exception when rolling back transaction
    catch (SystemException systemException) {
        throw new EJBException(systemException);
    }
    // throw EJBException indicating transaction failed
    throw new EJBException(exception);
} // end method transfer

Fig. 14.27 MoneyTransferEJB implementation of MoneyTransfer remote interface using bean-managed transaction demarcation (part 2 of 6).
// get balance of Account at BankABC
public double getBankABCBalance() throws EJBException
{
    // get balance of Account at BankABC
    try {
        // select balance for Account # 12345
        String select = "SELECT balance FROM Account " +
                        "WHERE accountID = 12345";

        PreparedStatement selectStatement =
            bankOneConnection.prepareStatement( select );

        ResultSet resultSet = selectStatement.executeQuery();

        // get first record in ResultSet and return balance
        if ( resultSet.next() )
            return resultSet.getDouble( "balance" );
        else
            throw new EJBException( "Account not found" );
    } // end try

    // handle exception when getting Account balance
    catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }

} // end method getBankABCBalance

// get balance of Account at BankXYZ
public double getBankXYZBalance() throws EJBException
{
    // get balance of Account at BankXYZ
    try {
        // select balance for Account # 54321
        String select = "SELECT balance FROM Account " +
                        "WHERE accountID = 54321";

        PreparedStatement selectStatement =
            bankTwoConnection.prepareStatement( select );

        ResultSet resultSet = selectStatement.executeQuery();

        // get first record in ResultSet and return balance
        if ( resultSet.next() )
            return resultSet.getDouble( "balance" );
        else
            throw new EJBException( "Account not found" );
    } // end try

Fig. 14.27 MoneyTransferEJB implementation of MoneyTransfer remote interface using bean-managed transaction demarcation (part 3 of 6).
// handle exception when getting Account balance
catch (SQLException sqlException) {
    throw new EJBException(sqlException);
}
} // end method getBankXYZBalance

// set SessionContext
public void setSessionContext(SessionContext context) throws EJBException {
    sessionContext = context;
    openDatabaseResources();
}

// create MoneyTransfer instance
public void ejbCreate() {}
Fig. 14.27  **MoneyTransferEJB** implementation of **MoneyTransfer** remote interface using bean-managed transaction demarcation (part 5 of 6).
Lines 48–51 withdraw the given transfer amount from an account in the BankABC database. Lines 53–56 deposit the given transfer amount in an account in the BankXYZ database. Both of these updates are part of the transaction begun on line 34 even though they are in separate databases. Line 59 commits the transaction to save the updates to each database.

Lines 65–79 catch any Exceptions thrown from lines 46–61. Line 69 invokes UserTransaction method rollback to undo any updates that were made within the transaction boundaries. This rollback ensures that if any critical part of method transfer failed, all of the changes made in both databases are undone to ensure the integrity of the data. Lines 73–75 catch a SystemException, which is thrown by UserTransaction method rollback if the rollback fails. Lines 74 and 78 throw EJBExceptions to aid in debugging the application.

Methods getBankABCBalance (lines 84–111) and getBankXYZBalance (lines 114–141) execute simple SQL SELECT statements to retrieve the balances of the accounts at each bank. Methods setSessionContext (lines 144–150) and ejbActivate (lines 168–171) invoke method openDatabaseResources (lines 202–250) to create Connections and PreparedStatements for each database for use throughout the lifetime of the MoneyTransfer EJB instance. Methods ejbRemove (lines 156–159) and ejbPassivate (lines 162–165) invoke method closeDatabaseResources (lines 174–199) to close the Connections and PreparedStatements.

### 14.4.3 Container-Managed Transaction Demarcation

Container-managed transaction demarcation allows the EJB developer to implement an EJB without specifying transaction boundaries. The EJB deployer provides transaction demarcation semantics declaratively when deploying the application.

MoneyTransferEJB (Fig. 14.28) implements the MoneyTransfer remote interface using container-managed transaction demarcation. Method transfer (lines 25–51) is similar to method transfer in Fig. 14.27. Note, however, that this version of method transfer does not declare any transaction boundaries, as this is now the responsibility...
of the EJB deployer. The EJB deployer specifies the transaction semantics using one of the six transaction types listed in Fig. 14.29.

```java
// MoneyTransferEJB.java
// MoneyTransferEJB is a stateless session EJB for transferring funds from an Account at BankABC to an Account at BankXYZ using container-managed transaction demarcation.
package com.deitel.advjhtp1.ejb.transactions.containermanaged;

// Java core libraries
import java.util.*;
import java.sql.*;

// Java standard extensions
import javax.ejb.*;
import javax.naming.*;
import javax.sql.*;

public class MoneyTransferEJB implements SessionBean {
    private SessionContext sessionContext;
    private Connection bankOneConnection;
    private Connection bankTwoConnection;
    private PreparedStatement withdrawalStatement;
    private PreparedStatement depositStatement;

    // transfer funds from BankABC to BankXYZ
    public void transfer( double amount ) throws EJBException {
        try {
            withdrawalStatement.setDouble( 1, amount );
            withdrawalStatement.executeUpdate();
            depositStatement.setDouble( 1, amount );
            depositStatement.executeUpdate();
        } // end try
        catch ( SQLException sqlException ) {
            // handle exception withdrawing and depositing
        }
    }
}
```

Fig. 14.28 MoneyTransferEJB implementation of MoneyTransfer remote interface using container-managed transaction demarcation (part 1 of 5).
// throw EJBException to indicate transfer failed
// and roll back container-managed transaction
throw new EJBException( sqlException );
}
} // end method transfer

// get balance of Account at BankABC
public double getBankABCBalance() throws EJBException {
    // get balance of Account at BankABC
    try {
        // select balance for Account # 12345
        String select = "SELECT balance FROM Account " + "WHERE accountID = 12345";
        PreparedStatement selectStatement =
            bankOneConnection.prepareStatement( select );
        ResultSet resultSet = selectStatement.executeQuery();
        // get first record in ResultSet and return balance
        if ( resultSet.next() )
            return resultSet.getDouble( "balance" );
        else
            throw new EJBException( "Account not found" );
    } // end try
    // handle exception when getting Account balance
    catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }
} // end method getBankABCBalance

// get balance of Account at BankXYZ
public double getBankXYZBalance() throws EJBException {
    // get balance of Account at BankXYZ
    try {
        // select balance for Account # 54321
        String select = "SELECT balance FROM Account " + "WHERE accountID = 54321";
        PreparedStatement selectStatement =
            bankTwoConnection.prepareStatement( select );
        ResultSet resultSet = selectStatement.executeQuery();

Fig. 14.28 MoneyTransferEJB implementation of MoneyTransfer remote interface using container-managed transaction demarcation (part 2 of 5).
```java
98     // get first record in ResultSet and return balance
99     if ( resultSet.next() )
100        return resultSet.getDouble( "balance" );
101     else
102        throw new EJBException( "Account not found" );
103
104 } // end try
105
106     // handle exception when getting Account balance
107     catch ( SQLException sqlException ) {
108        throw new EJBException( sqlException );
109     }
110
111 } // end method getBankXYZBalance
112
113     // set SessionContext
114     public void setSessionContext( SessionContext context )
115        throws EJBException
116     {
117        sessionContext = context;
118        openDatabaseResources();
119     }
120
121     // create MoneyTransfer instance
122     public void ejbCreate() {}
123
124     // remove MoneyTransfer instance
125     public void ejbRemove() throws EJBException
126     {
127        closeDatabaseResources();
128     }
129
130     // passivate MoneyTransfer instance
131     public void ejbPassivate() throws EJBException
132     {
133        closeDatabaseResources();
134     }
135
136     // activate MoneyTransfer instance
137     public void ejbActivate() throws EJBException
138     {
139        openDatabaseResources();
140     }
141
142     // close database Connections and PreparedStatements
143     private void closeDatabaseResources() throws EJBException
144     {
145        // close database resources
146        try {
147
148
```

Fig. 14.28 MoneyTransferEJB implementation of MoneyTransfer remote interface using container-managed transaction demarcation (part 3 of 5).
// close PreparedStatements
depositStatement.close();
depositStatement = null;

withdrawalStatement.close();
withdrawalStatement = null;

// close database Connections
bankOneConnection.close();
bankOneConnection = null;

bankTwoConnection.close();
bankTwoConnection = null;
}

// handle exception closing database connections
catch ( SQLException sqlException ) {
    throw new EJBException( sqlException );
}

} // end method closeDatabaseConnections

// open database Connections and create PreparedStatements
private void openDatabaseResources() throws EJBException {
    // look up the BankABC and BankXYZ DataSources and create
    // Connections for each
    try {
        Context initialContext = new InitialContext();

        // get DataSource reference from JNDI directory
        DataSource dataSource = ( DataSource )
                initialContext.lookup("java:comp/env/jdbc/BankABC" );

        // get Connection from DataSource
        bankOneConnection = dataSource.getConnection();

        dataSource = ( DataSource ) initialContext.lookup("java:comp/env/jdbc/BankXYZ" );

        bankTwoConnection = dataSource.getConnection();

        // prepare withdraw statement for account #12345 at
        // BankABC
        String withdrawal = "UPDATE Account SET balance = " +
                "balance - ? WHERE accountID = 12345";

        withdrawalStatement =
                bankOneConnection.prepareStatement( withdrawal );

        withdrawalStatement.setInt( 1, amount );
        withdrawalStatement.executeUpdate();

        withdrawalStatement =
                bankTwoConnection.prepareStatement( withdrawal );

        withdrawalStatement.setInt( 1, amount );
        withdrawalStatement.executeUpdate();

    } catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }

} // end method openDatabaseResources

/* MoneyTransferEJB */

// remote implementation of MoneyTransfer

Fig. 14.28 MoneyTransferEJB implementation of MoneyTransfer remote
interface using container-managed transaction demarcation (part 4 of 5).
Chapter 14  Session EJBs and Distributed Transactions

Line 66 throws an `EJBException` in response to any `SQLException` thrown from lines 29–41. The `EJBContainer` rolls back the current transaction when method `transfer` throws an `EJBException` (line 48).

Figure 14.29 lists the available transaction types for container-managed persistence. The deployer specifies the transaction type for each business method when deploying the application.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NotSupported</code></td>
<td>Method does not support transactions. The EJB container suspends the existing transaction context if the method is invoked within a transaction context.</td>
</tr>
<tr>
<td><code>Required</code></td>
<td>Method requires a transaction. The EJB container creates a new transaction if the method is invoked without an existing transaction context and commits the transaction at the end of the method.</td>
</tr>
<tr>
<td><code>Supports</code></td>
<td>Method supports transactions. The EJB container will not create a new transaction if the method is invoked without an existing transaction context, but will execute the method as part of an existing transaction context if one is available.</td>
</tr>
<tr>
<td><code>RequiresNew</code></td>
<td>Method requires a new transaction. The EJB container suspends the existing transaction context and starts a new transaction if the method is invoked as part of another transaction.</td>
</tr>
</tbody>
</table>

Fig. 14.28  MoneyTransferEJB implementation of MoneyTransfer remote interface using container-managed transaction demarcation (part 5 of 5).
MoneyTransferEJBClient (Fig. 14.30) provides a user interface for interacting with the MoneyTransfer EJB. Lines 24–26 declare JTextFields to display the account balances and accept a user-input transfer amount. Line 34 invokes method createMoneyTransfer to create a new MoneyTransfer EJB instance. Line 37 invokes method createGUI to create and lay out GUI components for the application. The GUI consists of JTextFields for displaying account balances and inputting the transfer amount and a JButton for transferring funds. Line 40 invokes method displayBalances to display the current account balances at BankABC and BankXYZ.

Method createMoneyTransfer (lines 47–80) uses the MoneyTransferHome interface to create a MoneyTransfer EJB instance. Line 51 creates an InitialContext for locating the MoneyTransfer EJB in the JNDI directory. Lines 54–59 invoke InitialContext method lookup to get a remote reference to the MoneyTransferHome interface. Line 62 creates a new MoneyTransfer EJB instance by invoking MoneyTransferHome method create.

Method getTransferButton (lines 108–142) creates a JButton to transfer funds from BankABC to BankXYZ. Lines 120–124 read the transfer amount from the user and invoke MoneyTransfer method transfer. Line 127 invokes method displayBalances to update the display with the new account balances.

---

### Transaction Types

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory</td>
<td>The method must execute in an existing transaction context. The EJB container throws a <strong>TransactionRequiredException</strong> if the method is invoked without a valid transaction context.</td>
</tr>
<tr>
<td>Never</td>
<td>The method must not execute in a transaction context. The EJB container throws a <strong>RemoteException</strong> if the method is invoked inside a transaction context.</td>
</tr>
</tbody>
</table>

---

**Fig. 14.29** Transaction types for container-managed transaction demarcation (part 2 of 2).
Chapter 14  Session EJBs and Distributed Transactions  893

import javax.ejb.*;
import javax.rmi.*;
import javax.naming.*;

// Deitel libraries
import com.deitel.advjhtp1.ejb.transactions.*;

public class MoneyTransferEJBClient extends JFrame {

    private MoneyTransfer moneyTransfer;
    private JTextField bankABCBalanceTextField;
    private JTextField bankXYZBalanceTextField;
    private JTextField transferAmountTextField;

    // MoneyTransferEJBClient constructor
    public MoneyTransferEJBClient( String JNDIName )
    {
        super( "MoneyTransferEJBClient" );

        // create MoneyTransfer EJB for transferring money
        createMoneyTransfer( JNDIName );

        // create and lay out GUI components
        createGUI();

        // display current balances at BankABC and BankXYZ
        displayBalances();

        setSize( 400, 300 );
        setVisible( true );
    }

    // create MoneyTransferEJB for transferring money
    private void createMoneyTransfer( String JNDIName )
    {
        // look up MoneyTransfer EJB using given JNDIName
        try {
            InitialContext context = new InitialContext();

            // lookup MoneyTransfer EJB
            Object homeObject = context.lookup( JNDIName );

            // get MoneyTransfer interface
            MoneyTransferHome moneyTransferHome =
                ( MoneyTransferHome ) PortableRemoteObject.narrow(
                homeObject, MoneyTransferHome.class );

            // create MathTool EJB instance
            moneyTransfer = moneyTransferHome.create();
        } // end try
    }
}

Fig. 14.30  MoneyTransferEJBClient for interacting with MoneyTransfer EJB (part 2 of 6).
65 // handle exception when looking up MoneyTransfer EJB
66 catch ( NamingException namingException ) {
67     namingException.printStackTrace();
68 }
69
70 // handle exception when looking up MoneyTransfer EJB
71 catch ( CreateException createException ) {
72     createException.printStackTrace();
73 }
74
75 // handle exception when looking up MoneyTransfer EJB
76 catch ( RemoteException remoteException ) {
77     remoteException.printStackTrace();
78 }
79 }
80 // end method createMoneyTransfer
81
82 // display balance in account at BankABC
83 private void displayBalances() {
84     try {
85         // get and display BankABC Account balance
86         double balance = moneyTransfer.getBankABCBalance();
87         bankABCBalanceTextField.setText(
88             String.valueOf( balance ) );
89         // get and display BankXYZ Account balance
90         balance = moneyTransfer.getBankXYZBalance();
91         bankXYZBalanceTextField.setText(
92             String.valueOf( balance ) );
93     }
94     catch ( RemoteException remoteException ) {
95         JOptionPane.showMessageDialog( this,
96             remoteException.getMessage() );
97     }
98 }
99 // end method displayBalances
100
101 // create button to transfer funds between accounts
102 private JButton getTransferButton() {
103     JButton transferButton = new JButton( "Transfer" );
104     transferButton.addActionListener(
105         new ActionListener() {
106             public void actionPerformed( ActionEvent event )
107             {

Fig. 14.30 MoneyTransferEJBClient for interacting with MoneyTransfer EJB (part 3 of 6).
try {

    // get transfer amount from JTextField
    double amount = Double.parseDouble(
        transferAmountTextField.getText() );

    // transfer money
    moneyTransfer.transfer( amount );

    // display new balances
    displayBalances();
}

// handle exception when transferring money
catch ( RemoteException remoteException ) {
    JOptionPane.showMessageDialog(
        MoneyTransferEJBClient.this,
        remoteException.getMessage() );
}

} // end method actionPerformed

return transferButton;

} // end method getTransferButton

// create and lay out GUI components
private void createGUI()
{

    // create JTextFields for user input and display
    bankABCBalanceTextField = new JTextField( 10 );
    bankABCBalanceTextField.setEditable( false );

    bankXYZBalanceTextField = new JTextField( 10 );
    bankXYZBalanceTextField.setEditable( false );

    transferAmountTextField = new JTextField( 10 );

    // create button to transfer between accounts
    JButton transferButton = getTransferButton();

    // layout user interface
    Container contentPane = getContentPane();
    contentPane.setLayout( new GridLayout( 3, 2 ) );

    contentPane.add( transferButton );
    contentPane.add( transferAmountTextField );
    contentPane.add( new JLabel( "Bank ABC Balance: " ) );
    contentPane.add( bankABCBalanceTextField );

Fig. 14.30 MoneyTransferEJBClient for interacting with MoneyTransfer EJB (part 4 of 6).
contentPane.add( new JLabel( "Bank XYZ Balance: " ) );
contentPane.add( bankXYZBalanceTextField );

} // end method createGUI

// get WindowListener for exiting application
private WindowListener getWindowListener()
{
    // remove MoneyTransfer EJB when user exits application
    return new WindowAdapter() {
        public void windowClosing( WindowEvent event ) {
            // remove MoneyTransfer EJB
            try {
                moneyTransfer.remove();
            }
            catch ( RemoveException removeException ) {
                removeException.printStackTrace();
                System.exit( 1 );
            }
            catch ( RemoteException remoteException ) {
                remoteException.printStackTrace();
                System.exit( 1 );
            }
            System.exit( 0 );
        } // end method windowClosing
    } // end method getWindowListener

    // execute application
    public static void main( String[] args )
    {
        // ensure user provided JNDI name for MoneyTransfer EJB
        if ( args.length != 1 )
            System.err.println("Usage: java MoneyTransferEJBClient <JNDI Name>庭");
        else
            new MoneyTransferEJBClient( args[ 0 ] );
    }

Fig. 14.30 MoneyTransferEJBClient for interacting with MoneyTransfer EJB (part 5 of 6).
Figure 14.30 shows three screen captures of **MoneyTransferEJBClient**. Enter any legal value in the text field and press the **Transfer** button. This causes the EJB to update the database entries. The client then updates its GUI to reflect the new account balances.

**Fig. 14.30** **MoneyTransferEJBClient** for interacting with **MoneyTransfer** EJB (part 6 of 6).
14.4.5 Deploying the MoneyTransfer EJB

The MoneyTransfer EJB accesses two databases that store account information. The examples use the Cloudscape DBMS introduced in Chapter 8, JDBC. Follow the directions in Section 8.5 to set up the databases for the MoneyTransfer EJB. The SQL script for the databases, transactions.sql, is on the CD-ROM that accompanies this book. If you have not already done so, please see Appendix E for instructions on integrating Cloudscape with the J2EE 1.2.1 reference implementation. Add the following

```
jdbc/BankABC|jdbc:cloudscape:rmi:BankABC;create=true;|jdbc/BankXYZ|jdbc:cloudscape:rmi:BankXYZ;create=true;
```

to the jdbc.datasources property in the file C:\j2sdkee1.2.1\config\default.properties.

Deploying each versions of the MoneyTransfer EJB is very similar to deploying the other session EJBs. When deploying the MoneyTransfer EJB, we must create resource references to the two databases in the New Enterprise Bean Wizard (Fig. 14.31). Click the Add button to add a new resource reference. Fill in the Coded Name and JNDI Name fields with jdbc/BankABC for the BankABC database (Fig. 14.32) and jdbc/BankXYZ for the BankXYZ database (Fig. 14.33). In the Transaction Management dialog of the New Enterprise Bean Wizard select Bean-Managed Transactions for the bean-managed version of MoneyTransferEJB (Fig. 14.34) or Container-Managed Transactions for the container-managed version of MoneyTransferEJB (Fig. 14.35). For the container-managed EJB, select a transaction type for each method (see Section 14.4.3 for information about transaction types). The rest of the deployment is the same as the deployment of the other session EJBs. Be sure to run the Cloudscape server before executing MoneyTransferEJBClient.

---

**Fig. 14.31** Resource References dialog of New Enterprise Bean Wizard.
Fig. 14.32 Add Resource Reference for BankABC.

Fig. 14.33 Add Resource Reference for BankXYZ.
14.5 Internet and World Wide Web Resources

java.sun.com/products/ejb
Sun’s Enterprise Java Bean home page. Contains articles, documentation and examples.

www.javaworld.com/javaworld/topicalindex/jw-ti-ejb.html
List of EJB-related articles from JavaWorld.

www.jguru.com/faq/home.jsp?topic=EJB
List of frequently asked questions on EJBs at jguru.com.

www.theserverside.com
TheServerSide.com is an online community for J2EE developers. There are forums, articles and other resources for building applications with J2EE.

SUMMARY

• Every EJB consists of a remote interface, a home interface and an EJB implementation.
• The remote interface declares business methods that clients of the EJB may invoke. The home interface provides `create` methods for creating new instances of the EJB, `finder` methods for finding instances of the EJB and `remove` methods for removing instances of the EJB. The EJB implementation defines the business methods declared in the remote interface and the `create`, `remove` and `finder` methods of the home interface.
• EJBs have a complex life cycle that is managed by an EJB container. The EJB container creates classes that implement the home and remote interfaces.

• The J2EE specification defines six roles for implementing enterprise systems. Each role is responsible for producing some part of an enterprise application.

• The remote interface for an EJB declares the business methods that clients of the EJB may invoke. The remote interface must extend interface `javax.ejb.EJBObject`.

• Each method of the remote interface is required to declare that it throws `java.rmi.RemoteException`. Each method also may throw application-specific exceptions—for example, an `IllegalArgumentException` if a provided argument does not meet certain criteria.

• The home interface for an EJB declares methods for creating, removing and finding EJB instances. The home interface must extend interface `javax.ejb.EJBHome`.

• Depending on the EJB type (i.e., session or entity), the container invokes EJB implementation methods that correspond to methods `create` and `remove` and `finder` methods of the home interface.

• The EJB implementation defines the business methods declared in the EJB remote interface and the `create`, `remove` and `finder` methods declared in the EJB home interface. The EJB implementation must also implement the methods of interface `javax.ejb.SessionBean` for session EJBs, or interface `javax.ejb.EntityBean` for entity EJBs.

• The EJB container manages the life cycle, client interactions and method invocations, transactions, security and exceptions of an EJB. Clients of an EJB do not interact directly with the EJB. When a client invokes a business method of the EJB’s remote interface, the invocation goes first to the EJB container, which then delegates the business method invocation to the EJB implementation.

• Session EJBs exist for the duration of a client’s session. Each session EJB instance is associated with a single client. Session EJBs can manipulate data in a database, but unlike entity EJBs, session EJBs are not persistent and do not represent database data directly.

• Stateful session EJBs maintain state information between business method invocations. For example, a stateful session EJB could maintain information about a customer’s shopping cart while the customer browses an online store.

• Interface `SessionContext` extends interface `EJBContext`, which provides methods for obtaining information about the EJB container.

• The EJB container invokes method `ejbCreate` when a client invokes a `create` method in the home interface. The EJB implementation must provide an `ejbCreate` method for each `create` method declared in the home interface. The `ejbCreate` methods must have the same number and types of arguments as their corresponding `create` methods.

• The EJB container invokes method `ejbRemove` in response to an invocation of method `remove` in the home interface.

• The EJB container invokes method `ejbPassivate` when it determines that the EJB is no longer needed in memory.

• The EJB container invokes method `ejbActivate` to restore an EJB instance that the container passivated previously. The EJB container activates an EJB instance if a client invokes a business method of that EJB instance.

• RMI-IIOP allows RMI objects to interact with CORBA components, which communicate using the `Internet Inter-Orb Protocol` (IIOP). CORBA is a language-independent framework for building distributed systems. To enable interoperability among EJBs and CORBA components, EJBs communicate using RMI-IIOP. We discuss CORBA and RMI-IIOP in detail in Chapter 22.

• Stateless session EJBs maintain no state information between business method invocations. As a result, any stateless session EJB instance can be used to respond to any client’s request. This improves the performance of stateless session EJBs over stateful session EJBs.
• The Java 2 Enterprise Edition supports distributed transactions. A distributed transaction is a transaction that is applied across multiple databases or across multiple EJB servers.

• J2EE supports two methods for defining transaction boundaries: bean-managed transaction demarcation and container-managed transaction demarcation. Bean-managed transaction demarcation requires the EJB developer to code the transaction boundaries manually in the EJBs, using the Java Transaction Services. Container-managed transaction demarcation allows the EJB deployer to specify transaction boundaries declaratively when deploying EJBs.

• Bean-managed transaction demarcation requires the EJB developer to code the transaction boundaries manually in the EJBs. Bean-managed transaction demarcation may be used only with session EJBs.

• Container-managed transaction demarcation allows the EJB developer to implement an EJB without specifying transaction boundaries. The EJB deployer provides transaction demarcation semantics declaratively when deploying the application.

**TERMINOLOGY**

application server
bean-managed transaction demarcation
business methods
container-managed transaction demarcation
CORBA (Common Object Request Broker Architecture)
create methods
distributed transaction
EJB container
EJB implementation
EJB server
ejbActivate method
EJBContext interface
ejbCreate methods
ejbPassivate method
ejbRemove method
Enterprise JavaBeans (EJBs)
entity EJB
home interface
IllegalArgumentException
Internet Inter-Orb Protocol (IIOP)
J2EE (Java 2 Enterprise Edition)
Java Transaction Services (JTS)
java.rmi.RemoteException
java:comp/env naming context
javax.ejb.EJBHome
javax.ejb.EJBObject
javax.ejb.SessionBean
JNDI (Java Naming and Directory Interface)
least recently used
naming context
remote interface of an EJB
remove methods
RMI-IIOP
session EJB
SessionFactory interface
stateful session EJB
stateless session EJB

**SELF-REVIEW EXERCISES**

14.1 What are the two main types of session EJBs? What is the primary difference between them?

14.2 What three Java objects must the EJB developer provide for each EJB?

14.3 What are the responsibilities of the EJB container?

14.4 How does a client get a remote reference to an EJB instance?

14.5 What types of transaction demarcation can EJBs use? What are the benefits of each type?

**ANSWERS TO SELF-REVIEW EXERCISES**

14.1 There are stateful session EJBs and stateless session EJBs. Stateful session EJBs maintain state information between business method invocations in a client’s session. Stateless session EJBs maintain no state information between business method invocations.
14.2 The EJB developer must provide a remote interface, a home interface and the EJB implementation.

14.3 The EJB container is responsible for managing the life cycle of the EJB. The EJB container creates classes to implement the home and remote interfaces, and delegates business method invocations to the developer-supplied EJB implementation. The EJB container also provides runtime resources, such as database connections and transactions, as well as life-cycle management.

14.4 A client looks up the EJB’s home interface in a JNDI directory. For session EJBs, the client then invokes one of the home interface’s create methods. For entity EJBs, the client can invoke one of the home interface’s create methods or finder methods.

14.5 EJBs can use either bean-managed or container-managed transaction demarcation. Bean-managed transaction demarcation allows the developer to have fine-grained control over transaction boundaries. Container-managed transaction demarcation simplifies the EJB implementation by allowing the EJB deployer to specify transaction boundaries declaratively at deployment time.

**EXERCISES**

14.1 Stateless session EJBs offer a performance advantage over stateful session beans. Convert the example of Fig. 14.3, Fig. 14.4, Fig. 14.5 and Fig. 14.6 from a stateful session EJB to a stateless session EJB.

14.2 Add a new recursive business method `power(base, exponent)` to the `MathTool` EJB (Fig. 14.21, Fig. 14.22, Fig. 14.23) that, when invoked, returns

```
base exponent
```

For example, `power(3, 4) = 3 * 3 * 3 * 3`. If the `exponent` is not an integer greater than or equal to 1, throw an `IllegalArgumentException`. [Hint: The recursion step would use the relationship

```
base exponent = base · base exponent - 1
```

and the terminating condition occurs when exponent is equal to 1 because

```
base 1 = base
```

Modify the client in Fig. 14.24 to enable the user to enter the `base` and `exponent`.]


Objectives

- To understand how entity EJBs represent persistent data.
- To understand synchronization issues between EJBs and database data.
- To understand the life-cycle of an entity EJB.
- To understand the advantages and disadvantages of container-managed and bean-managed persistence.

There is nothing more requisite in business than dispatch.
Joseph Addison

All meanings, we know, depend on the key of interpretation.
George Eliot

Event that are predestined require but little management. They manage themselves. They slip into place while we sleep, and suddenly we are aware that the thing we fear to attempt, is already accomplished.
Amelia Barr
Chapter 15 Entity EJBs

15.1 Introduction

A fundamental part of an enterprise application is the information tier, which maintains data for the application. In this chapter, we introduce entity EJBs, which enable developers to build object-based representations of information-tier data, such as data stored in a relational database. EJB containers provide advanced features that simplify developing entity EJBs. For example, based on information provided at deployment time (e.g., SQL queries), an entity EJB’s container can generate code automatically for storing and retrieving data represented by the EJB. For entity EJBs that represent more complex data (e.g., data stored in multiple database tables), the programmer can implement code for storing and retrieving the data manually.

In this chapter, we present two versions of an entity EJB that represents a company employee. The first version demonstrates an entity EJB that uses JDBC to persist data to a relational database. The second version takes advantage of the container’s ability to manage data storage and retrieval to simplify the EJB implementation. After completing this chapter, you will be able to build and deploy entity EJBs through which business-logic components, such as session EJBs (Chapter 14), can access data in the information tier.

15.2 Entity EJB Overview

Each entity EJB instance represents a particular unit of data, such as a record in a database table. There are two types of entity EJBs—those that use bean-managed persistence and those that use container-managed persistence. Entity EJBs that use bean-managed persistence must implement code for storing and retrieving data from the persistent data sources they represent. For example, an entity EJB that uses bean-managed persistence might use the JDBC API to store and retrieve data in a relational database. Entity EJBs that use container-managed persistence rely on the EJB container to implement the data-access calls to their persistent data sources. The deployer must supply information about the persistent data source when deploying the EJB.
Entity EJBs provide **create** methods for creating new EJB instances, **remove** methods for removing EJB instances and **finder** methods for finding EJB instances. For entity EJBs that represent information in a database, each **create** method performs **INSERT** operations to create new records in the database, and each **remove** method performs **DELETE** operations to remove records from the database. Each **finder** method locates entity EJB instances that conform to certain search criteria (e.g., using **SELECT** operations). We discuss each of these method types throughout the chapter.

### 15.3 Employee Entity EJB

In the following sections, we build an entity EJB that represents an **Employee**. We provide two implementations of the **Employee** EJB. The first implementation (Section 15.5) uses bean-managed persistence to store and retrieve **Employee** information in an underlying database. The second implementation (Section 15.6) uses container-managed persistence. Both of these implementations use the same **Employee** remote interface and **EmployeeHome** interface, which we present in Section 15.4.

We use the Cloudscape database for storing **Employee** data. To create the **Employee** database, run the SQL script **employee.sql** that is on the CD-ROM that accompanies the book. Please see Chapter 8, JDBC, for instructions on running SQL scripts in Cloudscape. To configure the J2EE reference implementation to use the **Employee** database, append the text

```
|jdbc/Employee|jdbc:cloudscape:rmi:Employee;create=true
```

to the end of the **jdbc.datasources** property in the J2EE **default.properties** configuration file.

### 15.4 Employee EJB Home and Remote Interfaces

The **Employee** remote interface (Fig. 15.1) provides methods for setting and getting **Employee** information. Note that interface **Employee** extends interface **EJBObject** (line 11). This is a requirement for all EJB remote interfaces. The **Employee** remote interface provides **set** and **get** methods for each **Employee** property, including the **socialSecurityNumber**, **firstName**, **lastName**, **title** and **salary**. There is no **set** method for property **employeeID** because **employeeID** is the primary key. Each **set** and **get** method throws a **RemoteException**. This is required of all methods in the remote interface.

![Fig. 15.1 Employee remote interface for setting and getting Employee information (part 1 of 2).](image-url)
An EJB instance represents a particular row in the corresponding database table. The home interface for an entity EJB represents the table as a whole. The home interface provides finder methods for locating particular rows in the table and create methods for inserting new records. Interface EmployeeHome (Fig. 15.2) provides finder method findByPrimaryKey (lines 15–16) to locate instances of the Employee EJB based on a primary key. The primary key for the Employee EJB is the employeeID. Method findByPrimaryKey throws a FinderException if the Employee with the given primaryKey cannot be found. Method create (lines 19–20) creates new instances of the Employee EJB. Method create throws a CreateException if there is a problem with creating the EJB instance.
Method \texttt{findByPrimaryKey} is one type of \textit{finder} method for entity EJBs. Every entity EJB must have a \texttt{findByPrimaryKey} method that takes the entity EJB’s primary-key class as an argument. Entity EJBs also can define additional \textit{finder} methods. A \textit{finder} method name must begin with \texttt{findBy} and should end with the name of property to be used as the search criteria. For example, a \textit{finder} method for finding Employee\s based on the title property would be named \texttt{findByTitle}. A \textit{finder} method for finding Employees within a certain salary range would be named \texttt{findBySalaryRange}.

### 15.5 Employee EJB with Bean-Managed Persistence

This section describes the Employee EJB with bean-managed persistence and deploying the EJB. This bean-managed implementation uses JDBC to store Employee data in an underlying database.

#### 15.5.1 Employee EJB Implementation

Figure 15.3 shows the Employee EJB implementation using bean-managed persistence. Class \texttt{EmployeeEJB} implements interface \texttt{EntityBean} (line 15). All entity EJB implementations must implement interface \texttt{EntityBean}. Line 17 declares an \texttt{EntityContext} reference for the EJB’s \texttt{EntityContext}. The \texttt{EntityContext} provides the EJB with information about the container in which the EJB is deployed. The \texttt{Connection} object (line 18) is the EJB’s \texttt{Connection} to the Employee database. Lines 20–25 declare \texttt{private} member variables that cache data retrieved from the database and updates from the client.
// EmployeeEJB.java
// EmployeeEJB is an entity EJB that uses bean-managed
// persistence to persist Employee data in a database.
package com.deitel.advjhtp1.ejb.entity.bmp;

// Java core libraries
import java.sql.*;
import java.rmi.RemoteException;

// Java standard extensions
import javax.ejb.*;
import javax.sql.*;
import javax.naming.*;

public class EmployeeEJB implements EntityBean {
    private EntityContext entityContext;
    private Connection connection;
    private Integer employeeID;
    private String socialSecurityNumber;
    private String firstName;
    private String lastName;
    private String title;
    private Double salary;

    // get Employee ID
    public Integer getEmployeeID() {
        return employeeID;
    }

    // set social security number
    public void setSocialSecurityNumber( String number ) {
        socialSecurityNumber = number;
    }

    // get social security number
    public String getSocialSecurityNumber() {
        return socialSecurityNumber;
    }

    // set first name
    public void setFirstName( String name ) {
        firstName = name;
    }
}

Fig. 15.3 EmployeeEJB implementation of Employee remote interface using bean-managed persistence (part 1 of 8).
    // get first name
    public String getFirstName()
    {
        return firstName;
    }

    // set last name
    public void setLastName( String name )
    {
        lastName = name;
    }

    // get last name
    public String getLastName()
    {
        return lastName;
    }

    // set title
    public void setTitle( String jobTitle )
    {
        title = jobTitle;
    }

    // get title
    public String getTitle()
    {
        return title;
    }

    // set salary
    public void setSalary( Double amount )
    {
        salary = amount;
    }

    // get salary
    public Double getSalary()
    {
        return salary;
    }

    // create new Employee
    public Integer ejbCreate( Integer primaryKey )
    throws CreateException
    {
        employeeID = primaryKey;

        // INSERT new Employee in database
        try {

        } catch (Exception e) {
            throw new CreateException(e.getMessage);
        }
    }

---

Fig. 15.3 EmployeeEJB implementation of Employee remote interface using bean-managed persistence (part 2 of 8).
// create INSERT statement
String insert = "INSERT INTO Employee " + 
    "( employeeID ) VALUES ( ? )";

// create PreparedStatement to perform INSERT
PreparedStatement insertStatement = 
    connection.prepareStatement( insert );

// set values for PreparedStatement
insertStatement.setInt( 1, employeeID.intValue() );

// execute INSERT and close PreparedStatement
insertStatement.executeUpdate();
insertStatement.close();
return employeeID;

// throw EJBException if INSERT fails
catch ( SQLException sqlException ) {
    throw new CreateException( sqlException.getMessage() );
}

// do post-creation tasks when creating new Employee
public void ejbPostCreate( Integer primaryKey ) {}

// remove Employee information from database
public void ejbRemove() throws RemoveException
{
    // DELETE Employee record
    try {

        // get primary key of Employee to be removed
        Integer primaryKey = 
            ( Integer ) entityContext.getPrimaryKey();

        // create DELETE statement
        String delete = "DELETE FROM Employee WHERE " + 
            "employeeID = ?";

        // create PreparedStatement to perform DELETE
        PreparedStatement deleteStatement = 
            connection.prepareStatement( delete );

        // set values for PreparedStatement
        deleteStatement.setInt( 1, primaryKey.intValue() );

        // execute DELETE and close PreparedStatement
        deleteStatement.executeUpdate();
        deleteStatement.close();
    }
}
// throw new EJBException if DELETE fails
catch ( SQLException sqlException ) {
    throw new RemoveException( sqlException.getMessage() );
}
} // end method ejbRemove

// store Employee information in database
public void ejbStore() throws EJBException {

    // UPDATE Employee record
    try {

        // get primary key for Employee to be updated
        Integer primaryKey =
            ( Integer ) entityContext.getPrimaryKey();

        // create UPDATE statement
        String update = "UPDATE Employee SET " +
            "socialSecurityNumber = ?, firstName = ?, " +
            "lastName = ?, title = ?, salary = ? " +
            "WHERE employeeID = ?";

        // create PreparedStatement to perform UPDATE
        PreparedStatement updateStatement =
            connection.prepareStatement( update );

        // set values in PreparedStatement
        updateStatement.setString( 1, socialSecurityNumber );
        updateStatement.setString( 2, firstName );
        updateStatement.setString( 3, lastName );
        updateStatement.setString( 4, title );
        updateStatement.setDouble( 5, salary.doubleValue() );
        updateStatement.setInt( 6, primaryKey.intValue() );

        // execute UPDATE and close PreparedStatement
        updateStatement.executeUpdate();
        updateStatement.close();
    }

    // throw EJBException if UPDATE fails
    catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }
} // end method ejbStore

// load Employee information from database
public void ejbLoad() throws EJBException {

    // get Employee record from Employee database table
    try {

        // EmployeeEJB implementation of Employee remote interface using
        bean-managed persistence (part 4 of 8).
// get primary key for Employee to be loaded
Integer primaryKey =
    ( Integer ) entityContext.getPrimaryKey();

// create SELECT statement
String select = "SELECT * FROM Employee WHERE " +
    "employeeID = ?";

// create PreparedStatement for SELECT
PreparedStatement selectStatement =
    connection.prepareStatement( select );

// set employeeID value in PreparedStatement
selectStatement.setInt( 1, primaryKey.intValue() );

// execute selectStatement
ResultSet resultSet = selectStatement.executeQuery();

// get Employee information from ResultSet and update
// local member variables to cache data
if ( resultSet.next() ) {
    // get employeeID
    employeeID = new Integer( resultSet.getInt( "employeeID" ) );

    // get social-security number
    socialSecurityNumber = resultSet.getString( "socialSecurityNumber" );

    // get first name
    firstName = resultSet.getString( "firstName" );

    // get last name
    lastName = resultSet.getString( "lastName" );

    // get job title
    title = resultSet.getString( "title" );

    // get salary
    salary = new Double( resultSet.getDouble( "salary" ) );
}

} // end try

else
    throw new EJBException( "No such employee." );

// close PreparedStatement
selectStatement.close();

// end try

Fig. 15.3  EmployeeEJB implementation of Employee remote interface using
bean-managed persistence (part 5 of 8).
// throw EJBException if SELECT fails
catch ( SQLException sqlException ) {
    throw new EJBException( sqlException );
}
} // end method ejbLoad

// find Employee using its primary key
public Integer ejbFindByPrimaryKey( Integer primaryKey )
    throws FinderException, EJBException
{
    // find Employee in database
    try {
        // create SELECT statement
        String select = "SELECT employeeID FROM Employee " +
            "WHERE employeeID = ?";

        // create PreparedStatement for SELECT
        PreparedStatement selectStatement =
            connection.prepareStatement( select );

        // set employeeID value in PreparedStatement
        selectStatement.setInt( 1, primaryKey.intValue() );

        // execute selectStatement
        ResultSet resultSet = selectStatement.executeQuery();

        // return primary key if SELECT returns a record
        if ( resultSet.next() ) {
            // close resultSet and selectStatement
            resultSet.close();
            selectStatement.close();

            return primaryKey;
        }

        // throw ObjectNotFoundException if SELECT produces
        // no records
        else
            throw new ObjectNotFoundException();
    }

    // throw EJBException if SELECT fails
    catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }
} // end method ejbFindByPrimaryKey

---

Fig. 15.3 EmployeeEJB implementation of Employee remote interface using bean-managed persistence (part 6 of 8).
// set EntityContext and create DataSource Connection
public void setEntityContext( EntityContext context ) throws EJBException {
    // set entityContext
    entityContext = context;

    // look up the Employee DataSource and create Connection
    try {
        InitialContext initialContext = new InitialContext();

        // get DataSource reference from JNDI directory
        DataSource dataSource = ( DataSource ) initialContext.lookup("java:comp/env/jdbc/Employee");

        // get Connection from DataSource
        connection = dataSource.getConnection();
    }

    // handle exception if DataSource not found in directory
    catch ( NamingException namingException ) {
        throw new EJBException( namingException );
    }

    // handle exception when getting Connection to DataSource
    catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }
}

// unset EntityContext
public void unsetEntityContext() throws EJBException {
    entityContext = null;

    // close DataSource Connection
    try {
        connection.close();
    }

    // throw EJBException if closing Connection fails
    catch ( SQLException sqlException ) {
        throw new EJBException( sqlException );
    }

    // prepare connection for reuse
    finally {
        connection = null;
    }
}

Fig. 15.3 EmployeeEJB implementation of Employee remote interface using bean-managed persistence (part 7 of 8).
Fig. 15.3 EmployeeEJB implementation of Employee remote interface using bean-managed persistence (part 8 of 8).

Lines 28–91 provide implementations of the set and get business methods declared in the Employee remote interface. When a client invokes interface EmployeeHome method create, the EJB container invokes method ejbCreate (lines 94–126). This EJB uses bean-managed persistence, so method ejbCreate must implement the appropriate logic to create a new Employee in the underlying database. Line 97 sets the employeeID member variable to the value of the primaryKey argument. Lines 103–111 create a PreparedStatement to INSERT the new Employee in the database. Line 111 sets the employeeID in the PreparedStatement, and line 114 inserts the new record by invoking PreparedStatement method executeUpdate. Line 115 closes the PreparedStatement, and line 117 returns the employeeID primary key. Lines 121–122 catch the SQLException that could be thrown when creating, executing or closing the PreparedStatement. An SQLException would indicate a problem inserting the record, so line 122 throws a CreateException to indicate that method ejbCreate could not create the Employee EJB instance.

Method ejbCreate declares an Integer return type. All ejbCreate methods in entity EJBs are required to return the EJB’s primary-key class. The primary key for the Employee table is an integer, so the primary-key class for the Employee EJB is java.lang.Integer. Most EJBs use a standard Java class (e.g., Integer or String) as their primary-key class. If a database table has a complex primary key (i.e., a primary key that consists of more than one field), the developer must define a custom primary-key class. For an example of a custom primary-key class, please see the Order-Product EJB example in Chapter 19.

The EJB container invokes method ejbPostCreate (line 127) after invoking method ejbCreate to perform any required tasks after creating the EJB instance. For example, method ejbPostCreate could change the format of the socialSecurityNumber to include dashes. No additional work is needed in this EJB, so line 127 is an empty implementation of method ejbPostCreate.

When a client invokes method remove in either the Employee remote interface or interface EmployeeHome, the EJB container invokes method ejbRemove (lines 130–159). To remove an entity EJB instance means to DELETE the associated database record. Lines 136–137 get the primary key for the current EJB instance from the entityContext object associated with the EJB. Lines 140–145 create a PreparedStatement to DELETE the Employee record from the database. Line 148 sets the employeeID primary key in
the `PreparedStatement`. Lines 151–152 execute and close the `PreparedStatement`. Lines 156–158 catch the `SQLException` that could be thrown when creating, executing or closing the `PreparedStatement`. If an `SQLException` is thrown, line 157 throws a `RemoveException` to indicate that removing the `Employee` EJB failed.

The EJB container invokes method `ejbStore` to save `Employee` data in the database. The EJB container determines the best time to update the database, so method `ejbStore` is only called by the container. Lines 168–169 get the primary key for the current EJB instance from the `entityContext`. Lines 172–187 create a `PreparedStatement` to UPDATE the `Employee` record in the database. Lines 190–191 execute and close the `PreparedStatement`. Lines 195–196 catch the `SQLException` that could be thrown when creating, executing or closing the `PreparedStatement`. Line 196 throws an `EJBException` to indicate the `ejbStore` method failed to update the data properly. Line 196 passes the `SQLException` to the `EJBException` constructor to aid in debugging the application.

The EJB container invokes method `ejbLoad` to load `Employee` data from the database and store the data in the EJB instance’s member variables. The EJB container determines the best time to load from the database, so method `ejbLoad` is only called by the container. Lines 207–208 get the primary key for the current `Employee` EJB instance from the EJB’s `EntityContext`. Lines 211–219 create a `PreparedStatement` to SELECT the `Employee` data from the database. Line 226 ensures that the `ResultSet` contains data, and lines 229–247 update the `private` member variables with the data from the `ResultSet`. Line 252 throws a new `EJBException` if the `ResultSet` is empty. Lines 260–262 catch an `SQLException` and throw a new `EJBException` to indicate the `ejbLoad` method failed to load the `Employee` information from the database.

When a client invokes `EmployeeHome` method `findByPrimaryKey`, the EJB container invokes method `ejbFindByPrimaryKey` (lines 266–306). Method `ejbFindByPrimaryKey` must take a single argument whose type is the EJB’s primary-key class. Lines 273–281 create a `PreparedStatement` to SELECT data from the database for the `Employee` with the given `employeeID` primary key. Lines 287–294 check on whether the `Employee` was found and return its primary key. If the `Employee` with the given primary key was not found in the database, lines 298–299 throw an `ObjectNotFoundException`. If there was an error in searching for the `Employee`, lines 303–305 catch an `SQLException` and throw a new `EJBException`.

The EJB container invokes method `setEntityContext` after the EJB instance is first created, but before the EJB instance is associated with a particular database record. The `EntityContext` argument provides methods for obtaining information about the EJB container in which the EJB instance executes. Line 313 sets the `entityContext` member variable so other methods can use the given `EntityContext` to discover information about the EJB container. There are some restrictions on what can be done in method `setEntityContext`, because when this method is invoked, the EJB instance is not yet associated with a particular database record. For example, the EJB must not invoke `EntityContext` method `getPrimaryKey`, because there is no primary key associated with the current EJB instance. Invoking `EntityContext` method `getPrimaryKey` would cause an `IllegalStateException`.

Method `setEntityContext` should allocate resources the EJB instance will need throughout its lifetime. This `Employee` EJB implementation uses bean-managed persis-
tence and therefore needs a database Connection to exchange data with the database throughout the EJB’s lifetime. Line 317 creates a new InitialContext that the EJB will use to look up the Employee database in the JNDI directory. Lines 320–322 invoke InitialContext method lookup to obtain a DataSource reference to the Employee database. J2EE applications use a special JNDI naming context called java:comp/env, which stands for “component environment.” There are subcontexts in the java:comp/env naming context for different types of objects. For example, EJBs are found in the ejb subcontext and databases are found in the jdbc subcontext. Line 322 uses the JNDI name java:comp/env/jdbc/Employee to locate the Employee database in the JNDI directory. Line 325 sets the connection member variable to a newly created database Connection. If the database with the given name cannot be found, lines 329–331 catch a NamingException, which is thrown by the InitialContext constructor or by method lookup.

Common Programming Error 15.1

The java:comp/env naming context is available only within a J2EE application (e.g., in a servlet, JSP or EJB). Using this naming context in a Java application or applet will result in a javax.naming.NamingException.

The EJB container invokes method unsetEntityContext (lines 340–358) when the EJB instance is no longer needed. Method unsetEntityContext should deallocate any resources the EJB instance used throughout its lifetime. Line 346 closes the Connection to the database. If there was an error closing the database, lines 350–352 catch an SQLException and throw an EJBException. Lines 355–357 set the Connection to null for later reuse.

The EJB container maintains a pool of inactive EJB instances that the container can associate with particular database records when needed. This pooling prevents the EJB container from incurring the overhead required to create a new EJB instance each time an EJB instance is needed. The EJB container invokes method ejbPassivate (lines 361–364) to place an active EJB back in the inactive pool. Line 363 sets the employeeID to null because a passivated EJB is no longer associated with any particular database record.

The EJB container invokes method ejbActivate (lines 367–371) to activate an EJB instance taken from the pool of inactive EJB instances. Line 369 sets the employeeID to the primaryKey value retrieved from the EJB’s entityContext to associate the EJB with the appropriate database record.

15.5.2 Employee EJB Deployment

Deploying entity EJBs is similar to deploying session EJBs (Chapter 14), but with the following changes. In the General dialog of the New Enterprise Bean Wizard, select the Entity radio button in the Bean Type field (Fig. 15.4). Next, select the type of persistence in the Entity Settings dialog. Click the Bean-Managed Persistence radio button for bean-managed persistence and enter the class name for the primary key in the Primary Key Class text field (Fig. 15.5). Finally, in the Resource References dialog, add a reference to the Employee database. The information for the reference must match that in Fig. 15.6. The rest of the deployment is the same as deploying a session EJB. For the Employee EJB’s JNDI name, we recommend you specify BMPEmployee for the bean-managed persistence version.
Fig. 15.4 General dialog of New Enterprise Bean Wizard.

Fig. 15.5 Bean-Managed Persistence selected in Entity Settings dialog.
Figure 15.7 shows an Employee EJB implementation that uses container-managed persistence to simplify the EJB implementation. Class EmployeeEJB implements interface EntityBean (line 12), indicating that this is an entity EJB. Line 14 declares an EntityContext reference to store the EntityContext associated with the EJB instance.

Lines 17–22 declare the container-managed fields for the Employee EJB. Container-managed fields are member variables that a container-managed persistence EJB instance uses to cache data retrieved from the database. The container-managed fields must have the same names as the fields in the corresponding database table. The container-managed fields also must be marked public so the EJB container can access them directly. The EJB container is responsible for synchronizing the container-managed fields with the database. When deploying the EJB, the deployer must provide the SQL statements for updating, inserting, deleting and retrieving data from the database. Lines 25–88 provide implementations of the methods declared in the Employee remote interface.

Method ejbCreate (lines 91–96) takes an Integer primaryKey argument and stores this primaryKey in the employeeID member variable (line 93). The EJB container executes the SQL \texttt{INSERT} statement (supplied by the deployer) after method \texttt{ejbCreate} completes. This SQL \texttt{INSERT} statement takes the current values of the container-managed fields in the EJB and inserts their values into the database. Note that, although method \texttt{ejbCreate} specifies \texttt{Integer} as its return type (line 91), line 95 returns \texttt{null}. According to the EJB specification, method \texttt{ejbCreate} must return the primary-key class type, which for the Employee EJB is \texttt{Integer}. However, the EJB container ignores the return value from \texttt{ejbCreate} when using container-managed per-
sistence, so we return null. The EJB specification requires an ejbPostCreate method for each ejbCreate method. Line 99 provides an empty implementation of method ejbPostCreate because this EJB requires no further initialization.

Methods setEntityContext (lines 102–105) and unsetEntityContext (lines 108–111) manage the EntityContext member variable. This EJB implementation does not have any resources it needs throughout the EJB lifetime, so no other work is done in either method.

```java
// EmployeeEJB.java
// EmployeeEJB is an entity EJB that uses container-managed
// persistence to persist Employee data in a database.
package com.deitel.advjhtp1.ejb.entity.cmp;

import java.rmi.RemoteException;
import javax.ejb.*;

public class EmployeeEJB implements EntityBean {
    private EntityContext entityContext;

    public Integer employeeID;
    public String socialSecurityNumber;
    public String firstName;
    public String lastName;
    public String title;
    public Double salary;

    public Integer getEmployeeID()
    {
        return employeeID;
    }

    public void setSocialSecurityNumber( String number )
    {
        socialSecurityNumber = number;
    }

    public String getSocialSecurityNumber()
    {
        return socialSecurityNumber;
    }
}
```

**Fig. 15.7** EmployeeEJB implementation of Employee remote interface using container-managed persistence (part 1 of 3).
```java
// set first name
public void setFirstName( String name )
{
    firstName = name;
}

// get first name
public String getFirstName()
{
    return firstName;
}

// set last name
public void setLastName( String name )
{
    lastName = name;
}

// get last name
public String getLastName()
{
    return lastName;
}

// set title
public void setTitle( String jobTitle )
{
    title = jobTitle;
}

// get title
public String getTitle()
{
    return title;
}

// set salary
public void setSalary( Double amount )
{
    salary = amount;
}

// get salary
public Double getSalary()
{
    return salary;
}

// create new Employee instance
public Integer ejbCreate( Integer primaryKey )
{
    employeeID = primaryKey;
}
```

Fig. 15.7 EmployeeEJB implementation of Employee remote interface using container-managed persistence (part 2 of 3).
Methods \texttt{ejbActivate} (lines 114–117) and \texttt{ejbPassivate} (lines 120–123) perform the same functions they did in the bean-managed persistence version of the \texttt{Employee} EJB. Method \texttt{ejbActivate} associates an EJB instance taken from the inactive pool with a particular database record, and method \texttt{ejbPassivate} disassociates the EJB instance from its database record.

Methods \texttt{ejbLoad} (line 126), \texttt{ejbStore} (line 129) and \texttt{ejbRemove} (line 132) also perform the same functions they did in the bean-managed persistence version of the \texttt{Employee} EJB. However with container-managed persistence, the deployer specifies the necessary \texttt{SELECT}, \texttt{UPDATE} and \texttt{DELETE} statements when deploying the application.
The EJB container executes these statements at runtime. These methods could perform further processing on the data if necessary. No further processing is necessary in this Employee EJB, so these methods are given empty implementations.

**Common Programming Error 15.2**

For entity EJBs using container-managed persistence, the EJB container loads database data and invokes method `ejbLoad` when the client first invokes a business method. However, the business method must be invoked in a transaction context.

**Performance Tip 15.1**

Entity EJBs using container-managed persistence might not perform as well as their bean-managed persistence counterparts, because all database data is loaded before the EJB container invokes method `ejbLoad`. Entity EJBs using bean-managed persistence can defer the loading of data from the database until the data is needed, which may enhance performance for EJBs with large amounts of data.

To deploy the container-managed persistence version of the Employee EJB, follow the instructions in Section 15.5.2. We recommend you use the JNDI Name `CMPEmployee` for the container-managed persistence version of the Employee EJB. In the Persistence Management dialog (Fig. 15.8), click the Container-Managed Persistence radio button and place check marks next to each container-managed field. Enter the class name of the primary key in the Primary Key Class text field and select the primary key field name from the Primary Key Field Name drop-down menu.

![Fig. 15.8 Container-Managed Persistence selected in Entity Settings dialog.](image-url)
15.7 Employee EJB Client

Figure 15.9 shows class EmployeeEJBClient for interacting with entity EJB Employee. The GUI provides buttons for finding, adding, updating and deleting Employee EJBs, and JTextFields for displaying Employee information. Lines 26–27 declare an EmployeeHome reference and an Employee reference for accessing the Employee EJB. The EmployeeEJBClient constructor takes a String argument containing the JNDI Name of the Employee EJB to load. This client works with both the bean-managed and container-managed persistence versions of the Employee EJB, because both versions use the same home and remote interfaces. The constructor uses the JNDIName argument to determine which version of the Employee EJB to use.

Line 47 creates a new InitialContext to look up the Employee EJB using the JNDIName argument. Lines 50–51 look up the EmployeeHome interface using InitialContext method lookup. Lines 54–56 use PortableRemoteObject method narrow to convert the Object reference returned by method lookup to an EmployeeHome reference. We will use this EmployeeHome reference throughout the program to create, find and remove Employee EJBs. If there is an error creating the InitialContext or looking up the EmployeeHome interface, lines 60–62 catch a NamingException.

Method getEmployeeID (lines 74–85) displays a JOptionPane to prompt the user for an employeeID. This employeeID can be used to find an existing Employee EJB instance or to create a new Employee EJB instance.

Method getEmployee (lines 88–119) uses method getEmployeeID on line 91 to prompt the user for the employeeID of the Employee the user would like to find. Lines 99–100 use EmployeeHome method findByPrimaryKey to locate the Employee EJB instance for the Employee with the given employeeID. Method findByPrimaryKey returns a remote reference to the Employee EJB instance. Line 103 invokes method setCurrentEmployee to display the Employee’s information to the user. If there was an error communicating with the Employee EJB, lines 107–111 catch a RemoteException. If there is no Employee with the given employeeID, lines 114–118 catch a FinderException.

Method addEmployee (lines 122–150) adds a new Employee to the database by creating a new Employee EJB. Line 126 invokes method getEmployeeID to prompt the user for an employeeID. Line 133 invokes EmployeeHome method create to create a new Employee EJB instance with the given employeeID. Line 136 invokes method setCurrentEmployee to display the Employee’s information to the user. Lines 140–143 catch a CreateException, which is thrown if there is an error creating the Employee EJB. For example, if the employeeID entered by the user is already in use by another Employee EJB, the EJB container will throw a DuplicateKeyException, which is a subclass of CreateException. If there is an error communicating with the Employee EJB, lines 146–149 catch a RemoteException.

Method updateEmployee (lines 153–184) uses the values provided in the JTextFields to update the current Employee’s information. Lines 159–175 invoke the Employee EJB set methods with values from each JTextField. If there is an error communicating with the Employee EJB, lines 180–183 catch a RemoteException.

Method deleteEmployee (lines 187–214) deletes the current Employee from the database by invoking Employee EJB method remove (line 191). Lines 194–199 set the
JTextFields in the user interface to the empty string. If there is an error communicating with the Employee EJB, lines 204–207 catch a RemoteException. If there is an error removing the Employee EJB, lines 210–213 catch a RemoveException.

Method setCurrentEmployee (lines 217–266) takes an Employee remote reference argument and updates the user interface with the information from this Employee EJB. Lines 224–258 retrieve values for each of the Employee EJB properties and update the JTextFields in the user interface. If there is an error communicating with the Employee EJB, lines 262–265 catch a RemoteException.

```java
// EmployeeEJBClient.java
// EmployeeEJBClient is a user interface for interacting with
// bean- and container-managed persistence Employee EJBs.
package com.deitel.advjhtp1.ejb.entity.client;

// Java core libraries
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import java.rmi.RemoteException;

// Java standard extensions
import javax.swing.*;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.*;

// Deitel libraries
import com.deitel.advjhtp1.ejb.entity.*;

public class EmployeeEJBClient extends JFrame {
    // variables for accessing EJBs
    private InitialContext initialContext;
    private EmployeeHome employeeHome;
    private Employee currentEmployee;

    // JTextFields for user input
    private JTextField employeeIDTextField;
    private JTextField socialSecurityTextField;
    private JTextField firstNameTextField;
    private JTextField lastNameTextField;
    private JTextField titleTextField;
    private JTextField salaryTextField;

    // BMPEmployeeEJBClient constructor
    public EmployeeEJBClient( String JNDIName )
    {
        super( "Employee EJB Client" );
    }
}
```

Fig. 15.9 EmployeeEJBClient for interacting with Employee EJB (part 1 of 9).
// create user interface
createGUI();

// get EmployeeHome reference for Employee EJB
try {
    initialContext = new InitialContext();

    // look up Employee EJB using given JNDI name
    Object homeObject =
        initialContext.lookup( JNDIName );

    // get EmployeeHome interface
    employeeHome = ( EmployeeHome )
        PortableRemoteObject.narrow(
            homeObject, EmployeeHome.class );
}

// handle exception when looking up Employee EJB
catch ( NamingException namingException ) {
    namingException.printStackTrace( System.err );
}

// close application when user closes window
setDefaultCloseOperation( EXIT_ON_CLOSE );

// set size of application window and make it visible
setSize( 600, 300 );
setVisible( true );

} // end EmployeeEJBClient constructor

// prompt user for employeeID
private Integer getEmployeeID() {
    String primaryKeyString = JOptionPane.showInputDialog( 
        this, "Please enter an employeeID" );

    // check if primaryKeyString is null, else return
    // Integer
    if ( primaryKeyString == null )
        return null;
    else
        return new Integer( primaryKeyString );
}

// get Employee reference for user-supplied employeeID
private void getEmployee() {
    // prompt user for employeeID and get Employee reference
    Integer employeeID = getEmployeeID();

```java
// return if employeeID is null
if ( employeeID == null )
    return;

try {
    // find Employee with given employeeID
    Employee employee =
        employeeHome.findByPrimaryKey( employeeID );

    // update display with current Employee
    setCurrentEmployee( employee );
}

// handle exception when finding Employee
catch ( RemoteException remoteException ) {
    JOptionPane.showMessageDialog(
        EmployeeEJBClient.this,
        remoteException.getMessage() );
}

// handle exception when finding Employee
catch ( FinderException finderException ) {
    JOptionPane.showMessageDialog(
        EmployeeEJBClient.this, "Employee not " +
        "found: " + finderException.getMessage() );
}
} // end method getEmployee

// add new Employee by creating new Employee EJB instance
private void addEmployee()
{
    // prompt user for employeeID and create Employee
    try {
        Integer employeeID = getEmployeeID();

        // return if employeeID null
        if ( employeeID == null )
            return;

        // create new Employee
        Employee employee = employeeHome.create( employeeID );

        // update display with new Employee
        setCurrentEmployee( employee );
    }

    // handle exception when creating Employee
    catch ( CreateException createException ) {
        JOptionPane.showMessageDialog( this, createException.getMessage() );
    }
}
```

**Fig. 15.9** EmployeeEJBClient for interacting with Employee EJB (part 3 of 9).
// handle exception when creating Employee
    catch ( RemoteException remoteException ) {
        JOptionPane.showMessageDialog( this,
            remoteException.getMessage() );
    }
} // end method addEmployee

// update current Employee with user-supplied information
private void updateEmployee()
{
    // get information from JTextFields and update Employee
    try {
        // set Employee socialSecurityNumber
        currentEmployee.setSocialSecurityNumber(
            socialSecurityTextField.getText() );

        // set Employee firstName
        currentEmployee.setFirstName(
            firstNameTextField.getText() );

        // set Employee lastName
        currentEmployee.setLastName(
            lastNameTextField.getText() );

        // set Employee title
        currentEmployee.setTitle( titleTextField.getText() );

        // set Employee salary
        Double salary = new Double( salaryTextField.getText() );
        currentEmployee.setSalary( salary );
    } // end try

    // handle exception invoking Employee business methods
    catch ( RemoteException remoteException ) {
        JOptionPane.showMessageDialog( this,
            remoteException.getMessage() );
    }
} // end method updateEmployee

// delete current Employee
private void deleteEmployee()
{
    // remove current Employee EJB
    try {
        currentEmployee.remove();

        // clear JTextFields
        employeeIDTextField.setText( "" );
        socialSecurityTextField.setText( "" );
    } // end try

    // handle exception when creating Employee
    catch ( RemoteException remoteException ) {
        JOptionPane.showMessageDialog( this,
            remoteException.getMessage() );
    }
} // end method deleteEmployee

Fig. 15.9 EmployeeEJBClient for interacting with Employee EJB (part 4 of 9).
firstNameTextField.setText( "" );
lastNameTextField.setText( "" );
titleTextField.setText( "" );
salaryTextField.setText( "" );

} // end try

// handle exception when removing Employee
catch ( RemoteException remoteException ) {
    JOptionPane.showMessageDialog( this,
          remoteException.getMessage() );
}

// handle exception when removing Employee
catch ( RemoveException removeException ) {
    JOptionPane.showMessageDialog( this,
          removeException.getMessage() );
}

} // end method deleteEmployee

// update display with current Employee information
private void setCurrentEmployee( Employee employee )
{
    // get information for currentEmployee and update display
    try {
        currentEmployee = employee;

        // get the employeeID
        Integer employeeID = ( Integer )
          currentEmployee.getEmployeeID();

        // update display
        employeeIDTextField.setText( employeeID.toString() );

        // set socialSecurityNumber in display
        socialSecurityTextField.setText( currentEmployee.getSocialSecurityNumber() );

        // set firstName in display
        firstNameTextField.setText( currentEmployee.getFirstName() );

        // set lastName in display
        lastNameTextField.setText( currentEmployee.getLastName() );

        // set title in display
        titleTextField.setText( currentEmployee.getTitle() );

        // get Employee salary
        Double salary = (Double) currentEmployee.getSalary();

        Fig. 15.9 EmployeeEJBClient for interacting with Employee EJB (part 5 of 9).
248     // ensure salary is not null and update display
249     if ( salary != null ) {
250         NumberFormat dollarFormatter =
251             NumberFormat.getCurrencyInstance(
252                 Locale.US );
253
254         salaryTextField.setText( dollarFormatter.format( 
255             salary ) );
256     }
257     else
258         salaryTextField.setText( "" );
259 } // end try
260
261     // handle exception invoking Employee business methods
262     catch ( RemoteException remoteException ) {
263         JOptionPane.showMessageDialog( this,
264             remoteException.getMessage() );
265     }
266 } // end method setCurrentEmployee
267
268     // create the application GUI
269     private void createGUI()
270     {
271         // create JPanel for Employee form components
272         JPanel formPanel = new JPanel( new GridLayout( 6, 2 ) );
273
274         // create read-only JTextField for employeeID
275         employeeIDTextField = new JTextField();
276         employeeIDTextField.setEditable( false );
277         formPanel.add( new JLabel( "Employee ID" ) );
278         formPanel.add( employeeIDTextField );
279
280         // create JTextField and JLabel for social security #
281         socialSecurityTextField = new JTextField();
282         formPanel.add( new JLabel( "Social Security #" ) );
283         formPanel.add( socialSecurityTextField );
284
285         // create JTextField and JLabel for first name
286         firstNameTextField = new JTextField();
287         formPanel.add( new JLabel( "First Name" ) );
288         formPanel.add( firstNameTextField );
289
290         // create JTextField and JLabel for last name
291         lastNameTextField = new JTextField();
292         formPanel.add( new JLabel( "Last Name" ) );
293         formPanel.add( lastNameTextField );
294
295         // create JTextField and JLabel for job title
296         titleTextField = new JTextField();
297         formPanel.add( new JLabel( "Title" ) );
298         formPanel.add( titleTextField );
299
Fig. 15.9  EmployeeEJBClient for interacting with Employee EJB (part 6 of 9).
// create JTextField and JLabel for salary
salaryTextField = new JTextField();
formPanel.add( new JLabel( "Salary" ) );
formPanel.add( salaryTextField );

// add formPanel to the JFrame's contentPane
Container contentPane = getContentPane();
contentPane.add( formPanel, BorderLayout.CENTER );

// create JPanel for JButtons
JPanel employeeButtonPanel =
  new JPanel( new FlowLayout() );

// create JButton for adding Employees
JButton addButton = new JButton( "Add Employee" );
addButton.addActionListener(
  new ActionListener() {
    public void actionPerformed( ActionEvent event ) {
      addEmployee();
    }
  });
employeeButtonPanel.add( addButton );

// create JButton for saving Employee information
JButton saveButton = new JButton( "Save Changes" );
saveButton.addActionListener(
  new ActionListener() {
    public void actionPerformed( ActionEvent event ) {
      updateEmployee();
    }
  });
employeeButtonPanel.add( saveButton );

// create JButton for deleting Employees
JButton deleteButton = new JButton( "Delete Employee" );
deleteButton.addActionListener(
  new ActionListener() {
    public void actionPerformed( ActionEvent event ) {
      deleteEmployee();
    }
  });
employeeButtonPanel.add( deleteButton );
// create JButton for finding existing Employees
JButton findButton = new JButton( "Find Employee" );
findButton.addActionListener( 
    new ActionListener() { 
    public void actionPerformed( ActionEvent event ) 
    { 
        getEmployee(); 
    } 
};
employeeButtonPanel.add( findButton );

// add employeeButtonPanel to JFrame's contentPane
contentPane.add( employeeButtonPanel, BorderLayout.NORTH );

} // end method createGUI

// execute application
public static void main( String[] args )
{
    // ensure user provided JNDI name for Employee EJB
    if ( args.length != 1 ) {
        System.err.println( 
            "Usage: java EmployeeEJBClient <JNDI Name>" );
        System.exit( 1 );
    } 

    // start application using provided JNDI name
    else
        new EmployeeEJBClient( args[ 0 ] );
}
Method `createGUI` (lines 269–369) builds the user interface for the `EmployeeEJBClient`. Lines 275–303 create `JTextField`s and `JLabel`s for each `Employee` property. Lines 314–363 create `JButton`s and associated `ActionListener`s to allow the user to add, update, delete and find `Employees`.

Method `main` (lines 372–384) requires that the user provide a command-line argument containing the JNDI name of the `Employee EJB` to use in the application. This command-line argument enables the user to specify the JNDI name of either the bean- or container-managed persistence version of the `Employee EJB` (i.e., `BMPEmployee` or `CMPEmployee`). Lines 376–377 print instructions to the user if a JNDI name is not provided. Line 383 creates a new instance of the `EmployeeEJBClient` using the given JNDI name argument. When executing the `EmployeeEJBClient`, be sure that the client JAR for the `Employee EJB` is in the `EmployeeEJBClient`’s `CLASSPATH`.

15.8 Internet and World Wide Web Resources

[Java Sun](http://java.sun.com/products/ejb/news.html)
This site provides articles and news related to Enterprise JavaBeans.

[The Server Side](http://www.theserverside.com/resources/pdf/ejbmatrix11.pdf)
This document (in PDF format) provides a quick reference for EJB interfaces, classes and methods.

[Developer](http://java.sun.com/products/ejb/docs.html)
The Enterprise JavaBeans specification home page provides documentation and specifications for EJBs.

[Developer](http://developer.java.sun.com/developer/technicalArticles/J2EE/blueprints)
This article at the Java Developer Connection (requires free registration) discusses techniques for ensuring J2EE applications are portable across application servers from different vendors.

[Java Sun J2EE Blueprints](http://java.sun.com/j2ee/blueprints)
The J2EE Blueprints discuss proven strategies for building enterprise applications with J2EE. Included are technical articles that discuss the blueprints and code examples that implement the blueprints in real systems.
**SUMMARY**

- Entity EJBs provide an object-oriented representation of persistent data, such as data stored in a relational database.
- Entity EJBs that use bean-managed persistence require the developer to implement code for storing and retrieving data from the persistent data sources the EJB represents.
- Entity EJBs use container-managed persistence rely on the EJB container to implement the data-access calls to their persistent data sources. The deployer must supply information about the persistent data source when deploying the EJB.
- Entity EJBs provide `create` methods for creating new EJB instances, `remove` methods for removing EJB instances and `finder` methods for finding EJB instances. Each `create` method performs `INSERT` operations to create new records in the database. Each `remove` method performs `DELETE` operations to remove records from the database. Each `finder` method locates entity EJB instances that match certain criteria.
- Method `findByPrimaryKey` is one type of `finder` method for entity EJBs. Every entity EJB must have a `findByPrimaryKey` method that takes the entity EJB’s primary-key class as an argument. Entity EJBs can also define other `finder` methods. A `finder` method name must begin with `findBy` and should end with the name of property to be used as the search criteria.
- All entity EJB implementations must implement interface `EntityBean`.
- An `EntityContext` provides the EJB with information about the container in which the EJB is deployed.
- All `ejbCreate` methods in entity EJBs are required to return the EJB’s primary-key class. Most EJBs use a standard Java class (e.g., `Integer` or `String`) as their primary-key class.
- If a database table has a complex primary key (i.e., a primary key that consists of more than one field), the developer must define a custom primary-key class.
- The EJB container invokes method `ejbPostCreate` after invoking method `ejbCreate` to perform any required tasks after creating the EJB instance.
- To `remove` an entity EJB instance means to `DELETE` the associated database record. A `RemoveException` indicates that removing the EJB instance failed.
- There are some restrictions on what can be done in method `setEntityContext`, because when this method is invoked, the EJB instance is not yet associated with a particular database record. For example, the EJB must not invoke `EntityContext` method `getPrimaryKey`, because there is no primary key associated with the current EJB instance.
- Method `setEntityContext` should allocate resources the EJB instance will need throughout its lifetime (e.g., a database connection).
- J2EE applications use a special JNDI naming context called `java:comp/env`, which stands for “component environment.” EJBs are found in the `ejb` subcontext and databases are found in the `jdbc` subcontext.
- Container-managed fields are member variables that a container-managed persistence EJB instance uses to cache data retrieved from the database. The container-managed fields must have the same names as the fields in the corresponding database table. The container-managed fields also must be marked `public` so the EJB container can access them directly.

**TERMINOLOGY**

- application server
- complex primary key
- bean-managed persistence
- container-managed field
- business methods
- container-managed persistence
SELF-REVIEW EXERCISES

15.1 State which of the following are true and which are false. If false, explain why.
   a) Data associated with an entity EJB typically is stored in a relational database.
   b) The remote interface for an entity EJB represents the database table with which the EJB is associated.
   c) Entity EJBs that use bean-managed persistence require the deployer to specify SQL queries for inserting, updating, deleting and querying data from the database.
   d) The create methods in the home interface for an entity EJB insert new records into the underlying database.
   e) Entity EJBs should obtain necessary resources in method ejbCreate.
   f) Entity EJBs that use container-managed persistence must implement interface CMPEntityBean, whereas those that use bean-managed persistence must implement interface EntityBean.

15.2 Fill in the blanks in each of the following:
   a) For entity EJBs that use ______ persistence to represent data in a relational database, the ______ must specify SQL queries when deploying the EJB.
   b) For entity EJBs that use ______ persistence to represent data in a relational database, the ______ must implement code that synchronizes data with the database.
   c) Each create method in the home interface must have a corresponding ______ method in the EJB implementation.
   d) If an entity EJB has a complex primary key, the developer must provide a custom ______ that represents the complex primary key.

ANSWERS TO SELF-REVIEW EXERCISES

15.1  a) True. b) False. The remote interface specifies the business methods for the EJB. The home interface represents the database table. c) False. The developer must provide code for managing synchronization with the underlying database. d) True. e) False. Entity EJBs should obtain necessary resources in method setEntityContext. f) False. All entity EJBs must implement interface EntityBean.

15.2  a) container-managed, deployer. b) bean-managed persistence, developer. c) ejbCreate. d) primary-key class.

EXERCISES

15.3 What types of persistence can entity EJBs use? What are the benefits of each?

15.4 Create an entity EJB for the Titles table in the books database of Chapter 8. The entity EJB should use bean-managed persistence to synchronize its data members with the values in the database. Provide set and get methods for each field in the Titles table. Provide a create method that takes the ISBN and Title as arguments.

15.5 Modify the entity EJB you created in Exercise 15.4 to use container-managed persistence instead of bean-managed persistence.
Objectives

- To understand message-oriented middleware.
- To understand the point-to-point messaging model.
- To understand the publish/subscribe messaging model.
- To understand the difference between the two messaging models and when it is appropriate to use each.
- To understand how to use the Java Message Service (JMS) API to build messaging applications in Java.
- To introduce message-driven EJBs.

The next thing most like living one’s life over again seems to be a recollection of that life, and to make that recollection as durable as possible by putting it down in writing.

Benjamin Franklin

Sow good services: sweet remembrances will grow from them.

Anne Louise Germaine de Stael

…And many a message from the skies, …

Robert Burns
When creating enterprise applications, it often is useful for ordinarily uncoupled components of the applications to "talk" to each other. For instance, a book supplier’s clients need to communicate book orders to the supplier. One solution is to loosely couple the client’s buying application with the supplier’s ordering application through a messaging system, sometimes called message-oriented middleware (MOM). Messaging systems allow components to post messages for other components to read. There are two basic messaging-system models—point-to-point and publish/subscribe.

The point-to-point messaging model allows components to send messages to a message queue. In this model, the sender intends the messages for one message consumer—a target component that processes the received messages. When this target component connects to the queue to receive messages, the target component receives any messages not yet consumed. A message is consumed once the server sends the message to the target component. Note that in the point-to-point model, exactly one client consumes a message.

The publish/subscribe messaging model allows components to publish messages to a topic on a server. The server maintains various topics to which components can connect. Components interested in messages published to a particular topic can subscribe to that topic. When a publisher publishes a message to a given topic, current subscribers receive that message. Note that in the publish/subscribe model, unlike in the point-to-point model, zero or more subscribers consume each published message.
In both messaging models, a message consists of a header, properties (optional) and a body (also optional). The message header contains information, such as the message destination and the sending time. Message properties allow message receivers to select which types of messages they would like to receive; the sender of a message can set these properties. Message receivers use message selectors to filter out messages. Filtering is done on the server side; the server does not send those messages to the receiver.

**Message-driven beans** are Enterprise JavaBeans that support messaging. Like an EJB container can use any instance of a stateless session EJB to handle client requests, the EJB container for message-driven beans can use any instance of a message-driven bean to process incoming messages for a given queue or topic. Since the container may use any instance, message-driven beans cannot maintain state for a specific client. Using message-driven beans, a component can receive messages asynchronously, without tying up resources waiting for a message to arrive.

In this chapter, we explore the Java Message Service (JMS) API. JMS standardizes the API for enterprise messaging, and supports both the point-to-point and publish/subscribe messaging models. JMS provides five types of messages—BytesMessages, MapMessages, ObjectMessages, StreamMessages and TextMessages. There are several implementations of JMS available from various vendors. For more information on these vendors and JMS, please visit [java.sun.com/jms](http://java.sun.com/jms).

### 16.2 Installation and Configuration of J2EE 1.3

The Java 2 Enterprise Edition version 1.3 reference implementation provides support for the Java Message Service API. At the time of this writing, the beta 2 release of the J2EE 1.3 reference implementation was available. Note that J2EE 1.3 beta 2 supports only the Windows 2000, Windows NT 4.0, Solaris SPARC 7, Solaris SPARC 8 and RedHat Linux v. 6.1 operating systems. Also note that J2SE 1.3.1 is required for J2EE 1.3 beta 2. To install the J2EE 1.3 reference implementation, follow these steps:

2. Set the environment variables according to the values in Fig. 16.1.

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2EE_HOME</td>
<td>Directory in which J2EE 1.3 is installed (e.g., <code>C:\j2sdkee1.3</code>).</td>
</tr>
<tr>
<td>JAVA_HOME</td>
<td>Directory in which J2SE 1.3.1 is installed (e.g., <code>C:\jdk1.3.1</code>).</td>
</tr>
<tr>
<td>PATH</td>
<td>The existing PATH plus the J2EE 1.3 bin directory (e.g., <code>C:\j2sdkee1.3\bin</code>).</td>
</tr>
</tbody>
</table>

**Fig. 16.1** Setting environment variables for J2EE 1.3 installation.
16.3 Point-To-Point Messaging

The point-to-point messaging model (Fig. 16.2) allows clients to send messages to a message queue. A receiver connects to the queue to consume messages that have not been consumed. In general, messages in a queue are intended for exactly one client, so only one client connects as the receiver. If there is no receiver, the server maintains messages sent to the queue until a receiver connects and consumes those messages.

16.3.1 Voter Application: Overview

To demonstrate point-to-point messaging, we present an application that tallies votes for the voters’ favorite programming language. Class **Voter** (Section 16.3.2) sends votes as messages to the **Votes** queue. These messages are simple **TextMessage** objects (package javax.jms). The message body contains the candidate name. Class **VoteCollector** (Section 16.3.3) consumes the messages and tallies the votes. Class **VoteCollector** can connect to the **Votes** queue before or after messages have been sent. As additional votes arrive in the queue, the **VoteCollector** updates the tallies and displays the totals. Figure 16.3 provides a diagram overview of the application.

---

**Fig. 16.2** Point-to-point messaging model.

**Fig. 16.3** Voter application overview.
16.3.2 Voter Application: Sender Side

The sender side of the application consists of a single class—**Voter** (Fig. 16.4). Class **Voter** allows the user to select a programming language and sends a vote for the selected language to the **Votes** queue as a **TextMessage**.

```java
// Voter.java
// Voter is the GUI that allows the client to vote
// for a programming language. Voter sends the vote
// to the "Votes" queue as a TextMessage.
package com.deitel.advjhtp1.jms.voter;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;
import javax.jms.*;
import javax.naming.*;

public class Voter extends JFrame {

    private String selectedLanguage;

    // JMS variables
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    private QueueSender queueSender;

    // Voter constructor
    public Voter() {
        // lay out user interface
        super( "Voter" );

        Container container = getContentPane();
        container.setLayout( new BorderLayout() );

        JTextArea voteArea =
            new JTextArea( "Please vote for your\n" + "favorite programming language" );
        voteArea.setEditable( false );
        container.add( voteArea, BorderLayout.NORTH );

        JPanel languagesPanel = new JPanel();
        languagesPanel.setLayout( new GridLayout( 0, 1 ) );

        // add each language as its own JCheckBox
        // ButtonGroup ensures exactly one language selected
        ButtonGroup languagesGroup = new ButtonGroup();
        CheckBoxHandler checkBoxHandler = new CheckBoxHandler();
```

Fig. 16.4 **Voter** class submits votes as messages to queue (part 1 of 4).
String languages[] =
    { "C", "C++", "Java", "Lisp", "Python" };
selectedLanguage = "";

// create JCheckBox for each language
// and add to ButtonGroup and JPanel
for ( int i = 0; i < languages.length; i++ ) {
    JCheckBox checkBox = new JCheckBox( languages[ i ] );
    checkBox.addItemListener( checkBoxHandler );
    languagesPanel.add( checkBox );
    languagesGroup.add( checkBox );
}

container.add( languagesPanel, BorderLayout.CENTER );

// create button to submit vote
JButton submitButton = new JButton( "Submit vote!" );
container.add( submitButton, BorderLayout.SOUTH );

// invoke method submitVote when submitButton clicked
submitButton.addActionListener ( new ActionListener() {

    public void actionPerformed ( ActionEvent event ) {
        submitVote();
    }
});

// invoke method quit when window closed
addWindowListener(
    new WindowAdapter() {

        public void windowClosing( WindowEvent event ) {
            quit();
        }
    });

// connect to message queue
try {

    // create JNDI context
    Context jndiContext = new InitialContext();

    // retrieve queue connection factory and
    // queue from JNDI context
    QueueConnectionFactory queueConnectionFactory =
        ( QueueConnectionFactory ) jndiContext.lookup( "VOTE_FACTORY" );
    Queue queue = ( Queue ) jndiContext.lookup( "Votes" );

Fig. 16.4 Voter class submits votes as messages to queue (part 2 of 4).
// create connection, session and sender
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession( false,
    Session.AUTO_ACKNOWLEDGE );
queueSender = queueSession.createSender( queue );
}

// process Naming exception from JNDI context
catch ( NamingException namingException ) {
namingException.printStackTrace();
System.exit( 1 );
}

// process JMS exception from queue connection or session
catch ( JMSException jmsException ) {
jmsException.printStackTrace();
System.exit( 1 );
}
// end Voter constructor

// submit selected vote to "Votes" queue as TextMessage
public void submitVote() {

    if ( selectedLanguage != "" ) {

        // create text message containing selected language
        try {
            TextMessage voteMessage =
                queueSession.createTextMessage();
            voteMessage.setText( selectedLanguage );

            // send the message to the queue
            queueSender.send( voteMessage );
        }

        // process JMS exception
        catch ( JMSException jmsException ) {
            jmsException.printStackTrace();
        }
    }
// end method submitVote

// close client application
public void quit() {
    if ( queueConnection != null ) {

        // process JMS exception
        catch ( JMSException jmsException ) {
            jmsException.printStackTrace();
        }
    }
}

Fig. 16.4  Voter class submits votes as messages to queue (part 3 of 4).
// close queue connection if it exists
try {
    queueConnection.close();
}

// process JMS exception
catch ( JMSException jmsException ) {
    jmsException.printStackTrace();
}
System.exit( 0 );

// launch Voter application
public static void main( String args[] )
{
    Voter voter = new Voter();
    voter.pack();
    voter.setVisible( true );
}

// CheckBoxHandler handles event when checkbox checked
private class CheckBoxHandler implements ItemListener {

    // checkbox event
    public void itemStateChanged( ItemEvent event )
    {
        // update selectedLanguage
        JCheckBox source = ( JCheckBox ) event.getSource();
        selectedLanguage = source.getText();
    }
}

Fig. 16.4   Voter class submits votes as messages to queue (part 4 of 4).

Line 13 imports package javax.jms, which contains JMS API classes and interfaces. Lines 29–86 set up the GUI for the Voter client. Lines 53–58 create JCheckBox objects to allow the client to vote. Note that these are added to a ButtonGroup (lines 45–58) so that the user can select only one candidate.

Lines 89–120 set up JMS connections. Line 92 creates the JNDI context, from which the program retrieves a QueueConnectionFactory and Queue, identified by VOTE_FACTORY and Votes, respectively. Note that the server administrator must create the queue connection factory and queue (Section 16.3.4). A queue connection factory allows the client to create a QueueConnection (lines 102–103). The QueueConnection creates a QueueSession (lines 104–106). Finally, the QueueSession creates either a QueueSender or a QueueReceiver. Here, the QueueSession creates a QueueSender for the Votes queue (line 107); the QueueSender now can post messages to the queue.
When the user selects a language and clicks the Submit vote button (Fig. 16.5), method submitVote creates a TextMessage object and sets the text of the message to selectedLanguage (lines 131–133). Line 136 sends the message to the Votes queue.

The program invokes method quit (lines 148–165) when the user closes the application window. Lines 150–161 close the connection to the queue.

### 16.3.3 Voter Application: Receiver Side

In the Voter application example, a VoteCollector (Fig. 16.6) is the intended receiver of messages sent to the Votes queue. Class VoteCollector tallies and displays votes it receives from the queue. Note that the Voter can send messages to the queue before the VoteCollector has connected; once connected, the VoteCollector will receive these messages.
// VoteCollector constructor
public VoteCollector()
{
    super( "Vote Tallies" );

    Container container = getContentPane();

    // displayPanel will display tally results
    displayPanel = new JPanel();
    displayPanel.setLayout( new GridLayout( 0, 1 ) );
    container.add( new JScrollPane( displayPanel ) );

    // invoke method quit when window closed
    addWindowListener(
        new WindowAdapter() {
            public void windowClosing( WindowEvent event ) {
                quit();
            }
        });

    // connect to "Votes" queue
    try {
        // create JNDI context
        Context jndiContext = new InitialContext();

        // retrieve queue connection factory
        // and queue from JNDI context
        QueueConnectionFactory queueConnectionFactory =
            ( QueueConnectionFactory ) jndiContext.lookup( "VOTE_FACTORY" );
        Queue queue = ( Queue ) jndiContext.lookup( "Votes" );

        // create connection, session and receiver
        queueConnection =
            queueConnectionFactory.createQueueConnection();
        QueueSession queueSession =
            queueConnection.createQueueSession( false,
                                                Session.AUTO_ACKNOWLEDGE );
        QueueReceiver queueReceiver =
            queueSession.createReceiver( queue );

        // initialize and set message listener
        queueReceiver.setMessageListener( new VoteListener( this ) );

        // start connection
        queueConnection.start();
    }

Fig. 16.6 VoteCollector class retrieves and tallies votes (part 2 of 4).
// process Naming exception from JNDI context
catch ( NamingException namingException ) {
    namingException.printStackTrace();
    System.exit( 1 );
}

// process JMS exception from queue connection or session
catch ( JMSException jmsException ) {
    jmsException.printStackTrace();
    System.exit( 1 );
}

// end VoteCollector constructor

// add vote to corresponding tally
public void addVote( String vote ) {
    if ( tallies.containsKey( vote ) ) {
        // if vote already has corresponding tally
        TallyPanel tallyPanel =
            ( TallyPanel ) tallies.get( vote );
        tallyPanel.updateTally();
    }
    // add to GUI and tallies
    else {
        TallyPanel tallyPanel = new TallyPanel( vote, 1 );
        displayPanel.add( tallyPanel );
        tallies.put( vote, tallyPanel );
        validate();
    }
}

// quit application
public void quit() {
    if ( queueConnection != null ) {
        // close the queue connection if it exists
        try {
            queueConnection.close();
        }
        // process JMS exception
        catch ( JMSException jmsException ) {
            jmsException.printStackTrace();
            System.exit( 1 );
        }
    }
}
Class **VoteCollector** collects and tallies votes retrieved from the **Votes** queue. Variable **tallies** (line 19) is a **Map** from candidate names to corresponding **TallyPanel** objects (Fig. 16.9); it updates tallies when a new vote is received. Lines 27–45 lay out the GUI. Note that the GUI initially displays nothing as the **VoteCollector** has received no votes.

Lines 48–87 establish a connection to the **Votes** queue. Line 51 creates a JNDI context for retrieving the **QueueConnectionFactory** and **Queue** (lines 55–58). The **QueueConnectionFactory** allows the program to create the **QueueConnection** (lines 61–62). The **QueueConnection** creates the **QueueSession** (lines 63–65). The **QueueSession**, in turn, creates the **QueueReceiver** (lines 66–67), which receives votes from the **Queue**. Lines 70–71 create a new **VoteListener** (Fig. 16.8) and set the **VoteListener** as the message listener for the **QueueReceiver**. Line 74 starts the **queueConnection**; at this point, the **VoteListener** will process messages received from the **queue**.

Method **addVote** (lines 92–109) updates the tallies and display (Fig. 16.7). If there is a **TallyPanel** corresponding to the **vote**'s candidate, lines 97–99 increment the tally by invoking method **updateTally**. If there is no corresponding **TallyPanel**, lines 104–106 add a new entry to the **tallies Map**. The program invokes method **quit** (lines 112–131) when the client closes the application window. Lines 114–127 close the **queue-Connection**.

**Fig. 16.6** **VoteCollector** class retrieves and tallies votes (part 4 of 4).

```java
129   System.exit( 0 );
130
131   } // end method quit
132
133   // launch VoteCollector
134   public static void main( String args[] )
135   {
136       VoteCollector voteCollector = new VoteCollector();
137       voteCollector.setSize( 200, 200 );
138       voteCollector.setVisible( true );
139   }
140 }
```

**Fig. 16.7** **VoteCollector** tallies and displays votes.
Class **VoteListener** (Fig. 16.8) implements interface **MessageListener**. When the **QueueReceiver** receives a message, the **QueueReceiver**’s **message listener** processes the message. Interface **MessageListener** specifies the single method **onMessage** (lines 22–49), which the program invokes when a new message arrives. Line 29 checks that the message is of type **TextMessage**. If so, lines 30–31 retrieve the text of the message; line 32 then invokes method **addVote** of class **VoteCollector** to update the tallies.

```java
// VoteListener.java
// VoteListener is the message listener for the
// receiver of the "Votes" queue. It implements
// the specified onMessage method to update the
// GUI with the received vote.
package com.deitel.advjhtp1.jms.voter;

import javax.jms.*;

public class VoteListener implements MessageListener {

    private VoteCollector voteCollector;

    // VoteListener constructor
    public VoteListener( VoteCollector collector ) {
        voteCollector = collector;
    }

    // receive new message
    public void onMessage( Message message ) {
        TextMessage voteMessage;

        // retrieve and process message
        try {
            if ( message instanceof TextMessage ) {
                voteMessage = ( TextMessage ) message;
                String vote = voteMessage.getText();
                voteCollector.addVote( vote );

                System.out.println( "Received vote: " + vote );
            }
        }

        else {
            System.out.println( "Expecting " +
                                "TextMessage object, received " +
                                message.getClass().getName() );
        }
    }
}
```

Fig. 16.8 **VoteListener** class receives messages from the queue (part 1 of 2).
Class **TallyPanel** (Fig. 16.9) maintains and displays the **tally** for a vote candidate. Method **updateTally** (lines 37–41) increments the **tally** by one.
16.3.4 Voter Application: Configuring and Running

To run the application, issue the following commands at a command prompt:

1. Start J2EE server in a command window:
   
   ```
   j2ee -verbose
   ```

2. In a new command window, create the **Votes** queue:
   
   ```
   j2eeadmin -addJmsDestination Votes queue
   ```

3. Verify that the queue was created:
   
   ```
   j2eeadmin -listJmsDestination
   ```

4. Create the connection factory:
   
   ```
   j2eeadmin -addJmsFactory VOTE_FACTORY queue
   ```

5. Start **VoteCollector**:
   
   ```
   java -classpath %J2EE_HOME%\lib\j2ee.jar;.
   -Djms.properties=%J2EE_HOME%\config\jms_client.properties
   com.deitel.advjhtp1.jms.voter.VoteCollector
   ```

6. Start **Voter** in a new command window:
   
   ```
   java -classpath %J2EE_HOME%\lib\j2ee.jar;.
   -Djms.properties=%J2EE_HOME%\config\jms_client.properties
   com.deitel.advjhtp1.jms.voter.Voter
   ```

Once you have finished running the application, you can remove the connection factory with the command:

```
j2eeadmin -removeJmsFactory VOTE_FACTORY
```

To remove the topic, use the command:

```
j2eeadmin -removeJmsDestination Votes
```

To stop J2EE server, use the command:

```
j2ee -stop
```

16.4 Publish/Subscribe Messaging

The publish/subscribe messaging model (Fig. 16.10) allows multiple clients to connect to a topic on the server and send and receive messages. Once connected, the client may publish messages or subscribe to the topic. When a client publishes a message, the server sends the message to those clients that subscribe to the topic. A client may obtain two types of subscriptions—**nondurable** and **durable**. Nondurable subscriptions receive messages only while the subscriptions are active. Durable subscriptions, however, can receive messages...
while inactive—the server maintains messages sent to the topic while the subscription is inactive and sends those messages to the client when the client reactivates the subscription. Note that if a client specifies a message selector to filter the messages, the server maintains only those messages that satisfy the selector.

### 16.4.1 Weather Application: Overview

We present an example of publish/subscribe messaging that uses a nondurable subscription. The following application publishes messages to a *Weather* topic on the server. These messages contain weather information for various U. S. cities retrieved from the National Weather Service Travelers Forecast page ([iwin.nws.noaa.gov/iwin/us/traveler.html](http://iwin.nws.noaa.gov/iwin/us/traveler.html)). Class `WeatherPublisher` (Section 16.4.2) retrieves weather updates and publishes them as messages to the topic. Class `WeatherSubscriber` (Section 16.4.3), presents a graphical user interface that allows the user to select cities for which to display weather updates. Class `WeatherSubscriber` subscribes to the *Weather* topic and receives messages corresponding to the selected cities, using a message selector. Figure 16.11 provides an overview of the application.

---

**Fig. 16.10** Publish/subscribe messaging model.

**Fig. 16.11** Weather application overview.
16.4.2 Weather Application: Publisher Side

Class **WeatherPublisher** (Fig. 16.12) retrieves weather updates from the National Weather Service and publishes those updates as messages to the **Weather** topic. An **ObjectMessage** contains each city’s weather conditions. The **ObjectMessage**’s **String** property **City** specifies the corresponding city. The **WeatherSubscriber** uses the **City** property in its message selector.

```java
// WeatherPublisher.java
// WeatherPublisher retrieves weather conditions from the National Weather Service and publishes them to the Weather topic as ObjectMessages containing WeatherBeans. The city name is used in a String property "City" in the message header.
package com.deitel.advjhtp1.jms.weather;

// Java core packages
import java.io.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.jms.*;
import javax.naming.*;

// Deitel packages
import com.deitel.advjhtp1.rmi.weather.WeatherBean;

public class WeatherPublisher extends TimerTask {
    private BufferedReader in;
    private TopicConnection topicConnection;

    // WeatherPublisher constructor
    public WeatherPublisher()
    {
        // update weather conditions every minute
        Timer timer = new Timer();
        timer.scheduleAtFixedRate( this, 0, 60000 );

        // allow user to quit
        InputStreamReader inputStreamReader =
            new InputStreamReader( System.in );
        char answer = '0';

        // loop until user enters q or Q
        while ( !( ( answer == 'q' ) || ( answer == 'Q' ) ) ) {
            // read in character
            try {
                answer = ( char ) inputStreamReader.read();
            }
        }
    }
}
```

**Fig. 16.12** WeatherPublisher class publishes messages to Weather topic (part 1 of 5).
```java
// process IO exception
catch ( IOException ioException ) {
    ioException.printStackTrace();
    System.exit( 1 );
}
}

// close while

// close connections
try {

    // close topicConnection if it exists
    if ( topicConnection != null ) {
        topicConnection.close();
    }

    in.close(); // close connection to NWS Web server
    timer.cancel(); // stop timer
}

// process JMS exception from closing topic connection
catch ( JMSException jmsException ) {
    jmsException.printStackTrace();
    System.exit( 1 );
}

// process IO exception from closing connection
// to NWS Web server
catch ( IOException ioException ) {
    ioException.printStackTrace();
    System.exit( 1 );
}

System.exit( 0 );
}
} // end WeatherPublisher constructor

// get weather information from NWS
public void run()
{
    // connect to topic "Weather"
    try {
        System.out.println( "Update weather information..." );

        // create JNDI context
        Context jndiContext = new InitialContext();
        String topicName = "Weather";
```

**Fig. 16.12** WeatherPublisher class publishes messages to Weather topic (part 2 of 5).
// retrieve topic connection factory and topic
// from JNDI context
TopicConnectionFactory topicConnectionFactory =
    ( TopicConnectionFactory )
jndiContext.lookup( "WEATHER_FACTORY" );

Topic topic =
    ( Topic ) jndiContext.lookup( topicName );

// create connection, session, publisher and message
TopicSession topicSession =
    topicConnectionFactory.createTopicConnection();

TopicPublisher topicPublisher =
    topicSession.createPublisher( topic );

ObjectMessage message =
    topicSession.createObjectMessage();

// connect to National Weather Service
// and publish conditions to topic
// National Weather Service Travelers Forecast page
URL url = new URL(
    "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );

// set up text input stream to read Web page contents
in = new BufferedReader(
    new InputStreamReader( url.openStream() ) );

// helps determine starting point of data on Web page
String separator = "TAV12";

// locate separator string in Web page
while ( !in.readLine().startsWith( separator ) )    // do nothing
    ;

// strings representing headers on Travelers Forecast
// Web page for daytime and nighttime weather
String dayHeader =
    "CITY            WEA     HI/LO   WEA     HI/LO";

String nightHeader =
    "CITY            WEA     LO/HI   WEA     LO/HI";

String inputLine = ";

Fig. 16.12 WeatherPublisher class publishes messages to Weather topic (part 3 of 5).
// locate header that begins weather information
do {
    inputLine = in.readLine();
}

while ( !inputLine.equals( dayHeader ) &&
    !inputLine.equals( nightHeader ) );

// create WeatherBean objects for each city's data
// publish to Weather topic using city as message's type
inputLine = in.readLine();  // get first city's info

// the portion of inputLine containing relevant data is
// 28 characters long. If the line length is not at
// least 28 characters long, done processing data.
while ( inputLine.length() > 28 ) {

    // create WeatherBean object for city
    // first 16 characters are city name
    // next six characters are weather description
    // next six characters are HI/LO temperature
    WeatherBean weather = new WeatherBean(
        inputLine.substring( 0, 16 ).trim(),
        inputLine.substring( 16, 22 ).trim(),
        inputLine.substring( 23, 29 ).trim() );

    // publish WeatherBean object with city name
    // as a message property,
    // used for selection by clients
    message.setObject( weather );
    message.setStringProperty( "City",
        weather.getCityName() );
    topicPublisher.publish( message );

    System.out.println( "published message for city: "
        + weather.getCityName() );

    inputLine = in.readLine();  // get next city's info
}

System.out.println( "Weather information updated." );

} // end try

// process Naming exception from JNDI context
catch ( NamingException namingException ) {
    namingException.printStackTrace();
    System.exit( 1 );
}

Fig. 16.12 WeatherPublisher class publishes messages to Weather topic (part 4 of 5).
Class **WeatherPublisher** publishes weather information for U.S. cities to the `Weather` topic on the server. Every minute, the `WeatherPublisher` obtains the weather information from the National Weather Service and publishes an individual message for each city’s weather. Figure 16.13 shows the server in progress. Method `run` (lines 83–213)—specified by class `TimerTask`—connects to the topic and publishes the messages. Line 90 creates the JNDI context, in which the `WeatherPublisher` looks up the `TopicConnectionFactory` and `Topic` (lines 95–100). The `TopicConnectionFactory`—which the server administrator must create—creates a `TopicConnection` (lines 103–104). The `TopicConnection` then creates a `TopicSession` (lines 106–108). Lines 110–111 obtain the `TopicPublisher` from the `TopicSession`. The `TopicSession` also creates an `ObjectMessage` (lines 113–114); the `ObjectMessage` will contain `WeatherBean` objects (see Fig. 13.3 in Chapter 13, Remote Method Invocation) that maintain the weather information for a given city. Lines 165–168 create the `WeatherBean` object with data from the National Weather Service. Method `setObject` of class `ObjectMessage` stores the `WeatherBean` object in the message. Lines 174–175 use the `setStringProperty` of class `Message` to set `String` prop-
property \textit{City} to the city name. The subscriber can use this property to filter the messages. Finally, the \texttt{topicPublisher} publishes the message to the \texttt{topic} (line 176).

### 16.4.3 Weather Application: Subscriber Side

Class \texttt{WeatherSubscriber} (Fig. 16.14) subscribes to the \texttt{Weather} topic to receive weather updates for selected cities. Class \texttt{WeatherSubscriber} creates a graphical user interface, which allows the user to choose cities, and displays the resulting weather conditions.

As with the \texttt{WeatherPublisher}, the \texttt{WeatherSubscriber} uses a JNDI context to obtain the \texttt{TopicConnectionFactory} and \texttt{Topic} (lines 55–65). The \texttt{TopicConnectionFactory} creates a \texttt{TopicConnection} (lines 68–69), which creates a \texttt{TopicSession} (lines 72–73). Line 76 initializes the message listener for the topic receiver (created in method \texttt{getWeather}) to a new instance of the \texttt{WeatherListener} class (Fig. 16.17). Lines 90–135 set up the user interface; the weather conditions are displayed in a \texttt{WeatherDisplay} (Fig. 16.18). Figure 16.15 shows the client GUI.

---

```java
// WeatherSubscriber.java
// WeatherSubscriber presents a GUI for the client to request
// weather conditions for various cities. The WeatherSubscriber
// retrieves the weather conditions from the Weather topic;
// each message body contains a WeatherBean object. The message
// header contains a String property "City," which allows
// the client to select the desired cities.
package com.deitel.advjhtp1.jms.weather;

import java.awt.*;
import java.awt.event.*;

// Java core packages
import java.awt.*);
import java.awt.event.*;
```

---

**Fig. 16.13** \texttt{WeatherPublisher} publishing weather update messages.

**Fig. 16.14** \texttt{WeatherSubscriber} class allows user to receive weather updates (part 1 of 6).
// Java extension packages
import javax.swing.*;
import javax.naming.*;
import javax.jms.*;

public class WeatherSubscriber extends JFrame {
    // GUI variables
    private WeatherDisplay weatherDisplay;
    private JList citiesList;

    // cities contains cities for which weather updates are available on "Weather" topic

    // JMS variables
    private TopicConnection topicConnection;
    private TopicSession topicSession;
    private Topic topic;
    private TopicSubscriber topicSubscriber;
    private WeatherListener topicListener;

    // WeatherSubscriber constructor
    public WeatherSubscriber() {
        super( "JMS WeatherSubscriber..." );
        weatherDisplay = new WeatherDisplay();
        
        // set up JNDI context and JMS connections
        try {
            // create JNDI context
            Context jndiContext = new InitialContext();

            // retrieve topic connection factory
            // from JNDI context
            TopicConnectionFactory topicConnectionFactory = ( TopicConnectionFactory ) jndiContext.lookup( "WEATHER_FACTORY" );

            // retrieve topic from JNDI context
            String topicName = "Weather";

            // ...
topic = ( Topic ) jndiContext.lookup( topicName );

// create topic connection
topicConnection =
    topicConnectionFactory.createTopicConnection();

// create topic session
topicSession = topicConnection.createTopicSession( false,
            Session.AUTO_ACKNOWLEDGE );

    // initialize listener
    topicListener = new WeatherListener( weatherDisplay );

    // process Naming exception from JNDI context
    catch ( NamingException namingException ) {
        namingException.printStackTrace();
    }

    // process JMS exceptions from topic connection or session
    catch ( JMSException jmsException ) {
        jmsException.printStackTrace();
    }

    // lay out user interface
    Container container = getContentPane();
    container.setLayout( new BorderLayout() );

    JPanel selectionPanel = new JPanel();
    selectionPanel.setLayout( new BorderLayout() );

    JLabel selectionLabel = new JLabel( "Select Cities" );
    selectionPanel.add( selectionLabel, BorderLayout.NORTH );

    // create list of cities for which users
    // can request weather updates
    citiesList = new JList( cities );
    selectionPanel.add( new JScrollPane( citiesList ),
            BorderLayout.CENTER );

    JButton getWeatherButton = new JButton( "Get Weather..." );
    selectionPanel.add( getWeatherButton, BorderLayout.SOUTH );

    // invoke method getWeather when getWeatherButton clicked
    getWeatherButton.addActionListener ( new ActionListener() {

        public void actionPerformed ( ActionEvent event )
        {
            getWeather();
        }
    });
117 }  
118 ); // end call to addActionListener
119 container.add( selectionPanel, BorderLayout.WEST );
120 container.add( weatherDisplay, BorderLayout.CENTER );
121
122 // invoke method quit when window closed
123 addWindowListener(
124     new WindowAdapter() {
125         public void windowClosing( WindowEvent event )
126         {
127             quit();
128         }
129     } ); // end call to addWindowListener
130 } // end WeatherSubscriber constructor
131
132 // get weather information for selected cities
133 public void getWeather()
134 {
135     // retrieve selected indices
136     int selectedIndices[] = citiesList.getSelectedIndices();
137     if ( selectedIndices.length > 0 ) {
138         // if topic subscriber exists, method has
139         // been called before
140         if ( topicSubscriber != null ) {
141             // close previous topic subscriber
142             try {
143                 topicSubscriber.close();
144             }
145             // process JMS exception
146             catch ( JMSException jmsException ) {
147                 jmsException.printStackTrace();
148             }
149             // clear previous cities from display
150             weatherDisplay.clearCities();
151         }
152         // create message selector to retrieve specified cities
153         StringBuffer messageSelector = new StringBuffer();
154         messageSelector.append(
155                 "City = '" + cities[ selectedIndices[ 0 ] ] + "," );
for ( int i = 1; i < selectedIndices.length; i++ ) {
    messageSelector.append( " OR City = " +
        cities[ selectedIndices[ i ] ] + "" );
}

// create topic subscriber and subscription
try {
    topicSubscriber = topicSession.createSubscriber(
        topic, messageSelector.toString(), false );
    topicSubscriber.setMessageListener( topicListener );
    topicConnection.start();
}

// process JMS exception
catch ( JMSException jmsException ) {
    jmsException.printStackTrace();
}

} // end if

} // end method getWeather

// quit WeatherSubscriber application
public void quit() {
    // close connection and subscription to topic
    try {
        // close topic subscriber
        if ( topicSubscriber != null ) {
            topicSubscriber.close();
        }
        // close topic connection
        topicConnection.close();
    }
    // process JMS exception
    catch ( JMSException jmsException ) {
        jmsException.printStackTrace();
        System.exit( 1 );
    }
    System.exit( 0 );
} // end method quit

Fig. 16.14 WeatherSubscriber class allows user to receive weather updates (part 5 of 6).
The program invokes method `getWeather` (lines 140–193) when the user clicks the **Get Weather** button. If the method has been called previously (i.e., the user has clicked the button before), lines 149–163 close the previous `TopicSubscriber` so that a new subscriber can filter in the selected cities with a message selector. When a client specifies a message selector for a subscription, the server will only send messages satisfying that filter to the client. Lines 166–173 create a `messageSelector` so that the weather conditions for the selected cities will be displayed. The `messageSelector` syntax is based on SQL92 (see the `Message javadoc` for more details). Lines 177–178 create the `TopicSubscriber` from `TopicSession`, passing the `Topic` and `messageSelector` as parameters. The third parameter, with value `false`, indicates that the subscriber can receive messages published by its own connection. Line 179 sets the message listener for the `TopicSubscriber` to `topicListener`. A `TopicSubscriber`’s message listener handles new messages when they arrive. Finally, line 180 starts the `TopicConnection`; once the connection has started, the `TopicSubscriber` will receive messages published to the topic. Figure 16.16 shows the application after receiving messages.

When the user closes the application window, lines 202–204 close the `TopicSubscriber` if it exists. Line 207 closes the `topicConnection`.

Class **WeatherListener** (Fig. 16.17) implements interface `MessageListener`. It therefore defines method `onMessage` (lines 26–57) to receive incoming messages. When a new message arrives, line 32 checks that it is of the appropriate message type—`ObjectMessage`. If so, method `getObject` gets the `WeatherBean` from the `ObjectMessage`. The `WeatherBean` object is then passed to the `WeatherDisplay`, which displays the corresponding information for the user. Figure 16.16 shows the application window once it has received a weather update.
Fig. 16.16 WeatherSubscriber having received updated weather conditions.

```java
// WeatherListener.java
// WeatherListener is the MessageListener for a subscription
// to the Weather topic. It implements the specified onMessage
// method to update the GUI with the corresponding city's
// weather.
package com.deitel.advjhtp1.jms.weather;

// Java extension packages
import javax.jms.*;
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.rmi.weather.WeatherBean;

public class WeatherListener implements MessageListener {

    private WeatherDisplay weatherDisplay;

    // WeatherListener constructor
    public WeatherListener( WeatherDisplay display )
    {
        weatherDisplay = display;
    }

    // receive new message
    public void onMessage( Message message )
    {
        // retrieve and process message
        try {
            // ensure Message is an ObjectMessage
            if ( message instanceof ObjectMessage ) {
                // get WeatherBean from ObjectMessage
                ObjectMessage objectMessage = ( ObjectMessage ) message;
            }
        }
    }
}
```

Fig. 16.17 WeatherListener class subscribes to Weather topic to receive weather forecasts (part 1 of 2).
Class `WeatherDisplay` (Fig. 16.18) displays `WeatherBean` objects in `JList weatherList`. The `weatherList` uses classes `WeatherListModel` and `WeatherCellRenderer` (see Fig. 13.5 and Fig. 13.6 in Chapter 13, Remote Method Invocation) to display `WeatherBean` objects. Method `addItem` (lines 47–65) adds the specified `WeatherBean` item to the display if the corresponding city is not displayed currently. If the city is in the display already, lines 56–58 remove the city’s previous `WeatherBean` object and add the updated `WeatherBean` object.

```java
WeatherBean weatherBean =
    (WeatherBean) objectMessage.getObject();

    // add WeatherBean to display
    weatherDisplay.addItem( weatherBean );

    } // end if
    else {
        System.out.println( "Expected ObjectMessage," +
            " but received " + message.getClass().getName() );
    }

} // end try

    // process JMS exception from message
    catch ( JMSException jmsException ) {
        jmsException.printStackTrace();
    }

} // end method onMessage
```

**Fig. 16.17** WeatherListener class subscribes to Weather topic to receive weather forecasts (part 2 of 2).

**Fig. 16.18** WeatherDisplay displays WeatherBeans in a JList using a WeatherCellRenderer (part 1 of 3).
public class WeatherDisplay extends JPanel {
    // WeatherListModel and Map for storing WeatherBeans
    private WeatherListModel weatherListModel;
    private Map weatherItems;

    // WeatherDisplay constructor
    public WeatherDisplay() {
        setLayout( new BorderLayout() );

        // add WeatherBean item to display
        public void addItem( WeatherBean weather ) {
            String city = weather.getCityName();

            // check whether city is already in display
            if ( weatherItems.containsKey( city ) ) {
                // if city is in Map, and therefore in display
                // remove previous WeatherBean object
                WeatherBean previousWeather =
                    ( WeatherBean ) weatherItems.remove( city );

                weatherListModel.remove( previousWeather );
            }

            // add WeatherBean to Map and WeatherListModel
            weatherListModel.add( weather );
            weatherItems.put( city, weather );
        }
    }
} // end WeatherDisplay constructor

Fig. 16.18 WeatherDisplay displays WeatherBeans in a JList using a WeatherCellRenderer (part 2 of 3).
### 16.4.4 Weather Application: Configuring and Running

To run the application, issue the following commands at a command prompt:

1. Start J2EE server in a command window:
   ```bash
   j2ee -verbose
   ```
2. In a new command window, create the `Weather` topic:
   ```bash
   j2eeadmin -addJmsDestination Weather topic
   ```
3. Verify that the topic was created:
   ```bash
   j2eeadmin -listJmsDestination
   ```
4. Create the connection factory:
   ```bash
   j2eeadmin -addJmsFactory WEATHER_FACTORY topic
   ```
5. Start `WeatherPublisher`:
   ```bash
   java -classpath %J2EE_HOME%/lib/j2ee.jar;.
   -Djms.properties=%J2EE_HOME%/config/jms_client.properties
   com.deitel.advjhtp1.jms.weather.WeatherPublisher
   ```
6. Start `WeatherSubscriber` in a new command window:
   ```bash
   java -classpath %J2EE_HOME%/lib/j2ee.jar;.
   -Djms.properties=%J2EE_HOME%/config/jms_client.properties
   com.deitel.advjhtp1.jms.weather.WeatherSubscriber
   ```

Once you have finished running the application, you can remove the connection factory with the command:

```bash
j2eeadmin -removeJmsFactory WEATHER_FACTORY
```

To remove the topic, use the command:

```bash
j2eeadmin -removeJmsDestination Weather
```

To stop J2EE server, use the command:

```bash
j2ee -stop
```
16.5 Message-Driven Enterprise JavaBeans

Message-driven EJBs, or message-driven beans, are a new type of Enterprise JavaBean available in Enterprise JavaBeans version 2.0, which is part of the Java 2 Enterprise Edition version 1.3.¹ Message-driven beans are capable of processing JMS messages posted to a queue or topic. When a message is received, the EJB container uses any available instance of a particular message-driven bean to process the message. This is similar to the way an EJB container will use any instance of a stateless session EJB to handle a client request. Since any message-driven EJB instance may be used, message-driven beans are not specific to particular client and must not maintain client state information. Note that any given EJB instance may process messages from multiple clients. Unlike session and entity beans (which require developers to provide home and remote interfaces), message-driven beans require that developers to provide only the bean implementation class.

16.5.1 Voter Application: Overview

This section presents an implementation of the Voter application from Section 16.3 using a message-driven bean to tally the votes posted to the Votes queue. Class Voter (Fig. 16.4), which posts the vote messages to the queue, remains exactly the same—a benefit of loosely-coupled applications. The sender simply sends messages to the queue regardless of the receiver’s implementation. The receiving end of the application (Section 16.5.2) is now a message-driven bean. Entity bean Candidate (Fig. 16.20, Fig. 16.21 and Fig. 16.22) represents a particular candidate for which users can vote. Entity bean Candidate stores the vote tallies in database Voting. Message-driven bean VoteCollectorEJB (Fig. 16.23) uses the Candidate EJB to update the tallies when the Votes queue receives a new vote message. Class TallyDisplay (Fig. 16.24) accesses the Candidate EJB to present a GUI with the current tallies from the database. Figure 16.19 provides an overview of the application.

¹ At the time of this writing, the Java 2 Enterprise Edition Specification, version 1.3, was a Proposed Final Draft in the Java Community Process.
16.5.2 Voter Application: Receiver Side

The Candidate entity EJB represents a particular candidate for which users can vote and the total number of votes the Candidate has received. Home interface Candidate-Home (Fig. 16.20) provides methods findByPrimaryKey (lines 15–16) and findAllCandidates (lines 19–20) to locate a particular Candidate or a Collection of all Candidates, respectively. Method create (lines 23–24) creates a new Candidate EJB with the given candidateName and zero votes.

Remote interface Candidate (Fig. 16.21) provides method incrementVoteCount to add a new vote for the Candidate. Method getVoteCount (line 18) returns the current vote tally for the Candidate. Method getCandidateName (line 21) returns the Candidate’s name.

```java
// CandidateHome.java
// CandidateHome is the home interface for the Candidate EJB.
package com.deitel.advjhtp1.jms.mdb;

// Java core libraries
import java.rmi.*;
import java.util.*;

// Java standard extensions
import javax.ejb.*;

public interface CandidateHome extends EJBHome {
    // find Candidate with given name
    public Candidate findByPrimaryKey( String candidateName )
            throws RemoteException, FinderException;

    // find all Candidates
    public Collection findAllCandidates()
            throws RemoteException, FinderException;

    // create new Candidate EJB
    public Candidate create( String candidateName )
            throws RemoteException, CreateException;
}
```

Fig. 16.20 CandidateHome interface for Candidate EJB.

```java
// Candidate.java
// Candidate is the remote interface for the Candidate EJB, which maintains a tally of votes.
package com.deitel.advjhtp1.jms.mdb;

// Java core libraries
import java.rmi.RemoteException;

Fig. 16.21 Candidate remote interface for Candidate EJB (part 1 of 2).
```
Class **CandidateEJB** (Fig. 16.22) implements the **Candidate** entity EJB. The **Candidate** EJB uses container-managed persistence for storing **Candidate** information in a database. Lines 17–18 declare container-managed fields **voteCount** and **name** for storing the **Candidate**’s total vote tally and name, respectively. Method **incrementVoteCount** (lines 21–25) increments the **Candidate**’s vote tally. Method **getName** (lines 34–37) returns the **Candidate**’s name. Method **ejbCreate** (lines 40–47) creates a new **Candidate** EJB, sets the **Candidate**’s **name** and initializes the **voteCount** to zero.

---

**Fig. 16.22** **CandidateEJB** class to maintain candidate tallies (part 1 of 3).
public Integer getVoteCount()
{
    return voteCount;
}

public String getCandidateName()
{
    return name;
}

public String ejbCreate( String candidateName )
    throws CreateException
{
    name = candidateName;
    voteCount = new Integer( 0 );
    return null;
}

public void ejbPostCreate( String candidateName ) {}
To handle incoming messages from the Votes queue, the container uses message-driven bean class VoteCollectorEJB (Fig. 16.23). When VoteCollectorEJB receives a new vote, the container invokes method onMessage (lines 21–48).

```
// VoteCollectorEJB.java
// VoteCollectorEJB is a MessageDriven EJB that tallies votes.
package com.deitel.advjhtp1.jms.mdb;

// Java core packages
import java.util.*;
import java.rmi.*;

// Java extension packages
import javax.ejb.*;
import javax.rmi.*;
import javax.jms.*;
import javax.naming.*;

public class VoteCollectorEJB
    implements MessageDrivenBean, MessageListener {

    private MessageDrivenContext messageDrivenContext;

    // receive new message
    public void onMessage( Message message )
    {
        TextMessage voteMessage;

        // retrieve and process message
        try {
            if ( message instanceof TextMessage )
            {
                voteMessage = ( TextMessage ) message;
                String vote = voteMessage.getText();
                countVote( vote );
                System.out.println( "Received vote: " + vote );
            } // end if
        } // end try
    }
}
```

Fig. 16.22 CandidateEJB class to maintain candidate tallies (part 3 of 3).

Fig. 16.23 VoteCollectorEJB class tallies votes from Votes queue (part 1 of 3).
else {
    System.out.println( "Expecting " +
        "TextMessage object, received " +
        message.getClass().getName() );
}

} // end try

// process JMS exception from message
catch ( JMSException jmsException ) {
    jmsException.printStackTrace();
}

// add vote to corresponding tally
private void countVote( String vote ) {
    // CandidateHome reference for finding/creating Candidates
    CandidateHome candidateHome = null;

    // find Candidate and increment vote count
    try {
        // look up Candidate EJB
        Context initialContext = new InitialContext();

        Object object = initialContext.lookup( "java:comp/env/ejb/Candidate" );

        candidateHome =
            ( CandidateHome ) PortableRemoteObject.narrow( object, CandidateHome.class );

        // find Candidate for whom the user voted
        Candidate candidate =
            candidateHome.findByPrimaryKey( vote );

        // increment Candidate's vote count
        candidate.incrementVoteCount();
    }

    // if Candidate not found, create new Candidate
    catch ( FinderException finderException ) {
        // create new Candidate and increment its vote count
        try {
            Candidate newCandidate = candidateHome.create( vote );
            newCandidate.incrementVoteCount();
        }
    }

    Fig. 16.23 VoteCollectorEJB class tallies votes from Votes queue (part 2 of 3).
974 Messaging with JMS Chapter 16

Class VoteCollectorEJB implements interfaces MessageDrivenBean and MessageListener. The container invokes method ejbCreate following instantiation of a new bean instance and method ejbRemove just before the instance is destroyed. Interface MessageDrivenBean specifies method setMessageDrivenContext. Note that interface MessageDrivenBean also specifies that the message-driven context should be stored as an instance variable (messageDrivenContext).

Upon receiving a message, the container invokes method onMessage, specified by interface MessageListener. Line 28 checks that the received Message is of type TextMessage. If so, line 31 invokes method countVote to count the received vote. Method countVote (lines 51–104) looks up the Candidate EJB (lines 60–67) and invokes method findByPrimaryKey of interface CandidateHome to locate the Candidate for which the user voted. If the Candidate is found, line 74 invokes method incrementVoteCount of interface Candidate to add a vote for the Candidate. If the Candidate is not found, lines 79–92 catch a FinderException.
Class **TallyDisplay** (Fig. 16.24) displays a snapshot of the candidates and corresponding tallies. Class **TallyDisplay** uses the **Candidate** EJB to retrieve the voting data. Lines 36–46 look up the **Candidate** EJB and retrieve a Collection of all Candidates.

For each **Candidate**, lines 51–58 create and add a new **TallyPanel** (Fig. 16.26), passing the candidate name and vote count as parameters to the **TallyPanel** constructor.
Figure 16.25 demonstrates the `TallyDisplay`. Note that the `TallyDisplay` displays only those votes already registered; the `TallyDisplay` does not update as new votes are received.

Class `TallyPanel` (Fig. 16.26) displays the name and tally for an individual candidate in a `JPanel`. 

```java
47    // add TallyPanel with candidate name and
48    // vote count for each candidate
49    Iterator iterator = candidates.iterator();
50    while ( iterator.hasNext() ) {
51        Candidate candidate = ( Candidate ) iterator.next();
52        // create TallyPanel for Candidate
53        TallyPanel tallyPanel =
54            new TallyPanel( candidate.getCandidateName(),
55                candidate.getVoteCount().intValue() );
56        displayPanel.add( tallyPanel );
57    }
58
59    } // end try
60
61    // handle exception finding Candidates
62    catch ( FinderException finderException ) {
63        finderException.printStackTrace();
64    }
65    // handle exception looking up Candidate EJB
66    catch ( NamingException namingException ) {
67        namingException.printStackTrace();
68    }
69    // handle exception communicating with Candidate
70    catch ( RemoteException remoteException ) {
71        remoteException.printStackTrace();
72    }
73
74    } // end TallyDisplay constructor
75
76    // launch TallyDisplay application
77    public static void main( String args[] )
78    {
79        TallyDisplay tallyDisplay = new TallyDisplay();
80        tallyDisplay.setDefaultCloseOperation( EXIT_ON_CLOSE );
81        tallyDisplay.pack();
82        tallyDisplay.setVisible( true );
83    }
84
85
```

**Fig. 16.24** `TallyDisplay` displays candidate tallies from database (part 2 of 2).
Fig. 16.25 **TallyDisplay** displays candidate tallies from database.

```java
// TallyPanel.java
// TallyPanel is the GUI component which displays
// the name and tally for a vote candidate.
package com.deitel.advjhtp1.jms.mdb;

// Java core packages
import java.awt.*;

// Java extension packages
import javax.swing.*;

class TallyPanel extends JPanel {

    private JLabel nameLabel;
    private JTextField tallyField;
    private String name;
    private int tally;

    // TallyPanel constructor
    public TallyPanel( String voteName, int voteTally )
    {
        name = voteName;
        tally = voteTally;

        nameLabel = new JLabel( name );
        tallyField =
            new JTextField( Integer.toString( tally ), 10 );
        tallyField.setEditable( false );
        tallyField.setBackground( Color.white );

        add( nameLabel );
        add( tallyField );
    }

} // end TallyPanel constructor
```

Fig. 16.26 **TallyPanel** class displays the name and tally for a candidate.
16.5.3 Voter Application: Configuring and Running

This section presents the steps needed to deploy and run the message-driven bean Voter application. Since the application relies on the Cloudscape database, the following lines must be added to the file resource.properties within the config directory under the main J2EE directory (e.g., C:\j2sdkee1.3\config\resource.properties):

```java
jdbcDataSource.5.name=jdbc/Voting
jdbcDataSource.5.url=jdbc:cloudscape:rmi:VotingDB;create=true
```

Note that there may be several jdbcDataSource entries in this properties file. In the above examples, you should replace the 5 with the number of the last jdbcDataSource entry plus 1. For example, if the last jdbcDataSource entry is number 3, you would specify jdbcDataSource.4.name and jdbcDataSource.4.url. Once you have added these lines, start Cloudscape. Then start the J2EE server, using the command

```bash
j2ee -verbose
```

In a new command window, create the queue and connection factory (note that these might still exist, if created in Section 16.3.4):

```bash
j2eeadmin -addJmsDestination Votes queue
j2eeadmin -listJmsDestination
j2eeadmin -addJmsFactory VOTE_FACTORY queue
```

To deploy the VoteCollector application, start

```bash
deploytool
```

Create a new application by selecting File -> New Application from the menu bar. In the dialog, click Browse and navigate to the directory above the com directory. Enter VoteCollectorApp.ear as the File name and click New Application. Then click OK.

Now, add the Candidate EJB by selecting New Enterprise Bean from the File menu. In the EJB JAR dialog, select the Create new EJB File in Application radio button, highlighting VoteCollectorApp from the resulting pull-down menu. Enter VoteCollectorJAR as the EJB Display Name. See Fig. 16.27 for details. Click the Edit button to add the class files. Inside the Edit dialog, specify the directory that contains the com.deitel.advjhtp1.packages structure as the Starting Directory. Navigate down the tree to the mdb directory (com/deitel/advjhtp1/jms/mdb) and Add Candidate.class, CandidateEJB.class and CandidateHome.class (Fig. 16.28). Click OK to exit the Edit dialog. Click Next to proceed to the General dialog.

In the General dialog, select the Entity radio button as the Bean Type. Select com.deitel.advjhtp1.jms.mdb.CandidateEJB from the pull-down menu for the Enterprise Bean Class. Enter Candidate as the Enterprise Bean Name. In the Remote Interfaces section, select com.deitel.advjhtp1.jms.mdb.CandidateEJB and com.deitel.advjhtp1.jms.mdb.Candidate as the Remote Home Interface and Remote Interface, respectively. (See Fig. 16.29 for details.) Click Next to proceed to the Entity Settings dialog.
Fig. 16.27  EJB JAR settings for VoteCollectorApp application.

Fig. 16.28  Add class files for Candidate EJB.
In the **Entity Settings** dialog, select radio button **Container managed persistence (1.0)**. Check both **voteCount** and **name**. Enter `java.lang.String` as the **Primary Key Class**, and select **name** from the pull-down menu under the **Primary Key Field**. See Fig. 16.30 for details. Click **Finish**.

---

**Fig. 16.29** General settings for **Candidate EJB**.

**Fig. 16.30** Entity settings for **Candidate EJB**.
In the main deploytool window, select **Candidate** and click the **Entity** tab (Fig. 16.31). Click **Deployment Settings**. In the resulting dialog, click **Database Settings**. Enter `jdbc/voting` as the **Database JNDI Name** (Fig. 16.32). Click **OK**. In the **Deployment Settings** dialog, click **Generate Default SQL**. A dialog will pop up, indicating **SQL Generation complete**; click **OK** (Fig. 16.33). In the **Deployment Settings** dialog, click **OK**. A warning dialog will appear that indicates there is no **WHERE** clause for method **findAllCandidates**; click **OK** to ignore the warning (Fig. 16.34).

![Fig. 16.31 Entity tab for Candidate EJB.](image1)

![Fig. 16.32 Database settings for Candidate EJB.](image2)
Now create the **VoteCollector** EJB by selecting **File -> New Enterprise Bean** from the menu bar. Click radio button **Add to Existing EJB File** and select **VoteCollectorJAR(VoteCollectorApp)** from the resulting pull-down menu (Fig. 16.35). Click **Edit** and add **VoteCollectorEJB.class** from tree structure `com/deitel/advjhtp1/jms/mdb` (Fig. 16.36). Click **OK** to exit the **Edit** dialog. Click **Next** to proceed to the **General** dialog.

In the **General** dialog (Fig. 16.37), select **Message-Driven Bean** as the **Bean Type**. Select `com.deitel.advjhtp1.jms.mdb.VoteCollectorEJB` as the **Enterprise Bean Class**. Enter **VoteCollector** as the **Enterprise Bean Name**. Click **Next** to proceed to the **Transaction Management** dialog.

In the **Transaction Management** dialog (Fig. 16.38), select **Container-Managed**. Verify **Transaction Attribute Required** for method **onMessage**. Click **Next** to proceed to the **Message-Driven Bean Settings** dialog.
Fig. 16.35  EJB JAR settings for **VoteCollector** EJB.

Fig. 16.36  Add class file for **VoteCollector** EJB.
In the **Message-Driven Bean** dialog, select **Queue** as the **Destination Type** (Fig. 16.39). Select **Votes** and **VOTE_FACTORY** from the pull-down menus for **Destination** and **Connection Factory**, respectively. Click **Next** to proceed to the **Environment Entries** dialog. Do not enter anything in this dialog; click **Next** again to proceed to the **Enterprise Bean References** dialog.

In the **Enterprise Bean References** dialog (Fig. 16.40), click **Add**. For the **Coded Name**, enter **ejb/Candidate**. Select **Entity** for **Type** and **Remote** for **Interfaces**. Enter **com.deitel.advjhtp1.jms.mdb.CandidateHome** and **com.deitel.advjhtp1.jms.mdb.Candidate** as the **Home Interface** and **Remote Interface**, respectively. Click **Next** again to proceed to the **Transaction Management** dialog.
Local/Remote Interface, respectively. Select radio button JNDI Name and enter Candidate in the corresponding text field. Click Finish.

In the main deploytool window, select VoteCollectorApp from the tree, and click the JNDI Names tab. Enter Candidate as the JNDI Name for Candidate. Verify that Votes is the JNDI Name for VoteCollector (Fig. 16.41).

![Fig. 16.39 Message-Driven Bean settings for VoteCollector EJB.](image1)

![Fig. 16.40 Enterprise Bean References for VoteCollector EJB.](image2)
Finally, deploy the application by selecting **Deploy** from the **Tools** menu. Select **VoteCollectorApp** as the **Object to Deploy**. In the dialog, check **Return Client JAR** (Fig. 16.42). Click **Next** and verify the JNDI names. Click **Next**, then **Finish**.

---

**Fig. 16.41** Setting JNDI names for **VoteCollectorApp**.

**Fig. 16.42** Deploying the **VoteCollector** application.
After deploying the **VoteCollector** application, run the **Voter** client application:

```
java -classpath %J2EE_HOME%/lib/j2ee.jar; .
-Djms.properties=%J2EE_HOME%/config/jms_client.properties
com.deitel.advjhtp1.jms.mdb.Voter
```

To view the current vote tallies, run **TallyDisplay** (note that the client JAR must be included in the classpath)

```
java -classpath %J2EE_HOME%/lib/j2ee.jar;VoteCollectorAppClient.jar; .
com.deitel.advjhtp1.jms.mdb.TallyDisplay
```

**SUMMARY**

- A messaging system loosely couples components.
- Messaging systems allow components to post messages for other components to read.
- There are two basic messaging system models—point-to-point and publish/subscribe. The point-to-point messaging model allows components to send messages to a message queue. A message consumer is a target component that processes the received messages.
- In the point-to-point model, exactly one client consumes a message; the server maintains messages that have not been consumed.
- The publish/subscribe messaging model allows components to publish messages to a topic. Components interested in messages published to a particular topic can subscribe to that topic.
- When a publisher publishes a message to a given topic, current subscribers receive that message.
- In the publish/subscribe model, zero or more subscribers consume a published message.
- A message consists of a header, properties (optional) and a body (also optional). The message header contains information, such as the message destination and the sending time.
- Message properties allow message receivers to select which types of messages they would like to receive; the sender of a message can set these properties. Message receivers use message selectors to filter out messages; filtering is done on the server side.
- Message-driven beans are a type of enterprise bean that integrate nicely with MOM.
- The EJB container can use any instance of a message-driven bean to process incoming messages to a given queue or topic. Using message-driven beans, a component can receive messages asynchronously.
- The Java Message Service (JMS) API. JMS standardizes enterprise messaging, providing APIs for both the point-to-point and publish/subscribe models.
- JMS provides five types of messages—**BytesMessages**, **MapMessages**, **ObjectMessages**, **StreamMessages** and **TextMessages**.
- The server administrator creates the connection factories, queues and topics.
- A **QueueConnectionFactory** allows the client to create a **QueueConnection**.
- A **QueueConnection** creates a **QueueSession**. A **QueueSession** creates either a **QueueSender** or a **QueueReceiver**.
- When the queue receiver or topic subscriber receives a message, the message listener processes the message.
- Interface **MessageListener** declares method **onMessage** that is invoked when a new message arrives.
- A client may obtain two types of subscriptions—nondurable and durable. Nondurable subscriptions receive messages only while the subscriptions are active.
• Durable subscriptions can receive messages while inactive—the server maintains messages sent to the topic while the subscription is inactive and sends them to the client when the subscription is reactivated. Note that, if a message selector is specified for the subscription to filter the messages, the server maintains only the messages that satisfy the selector.

• A `TopicConnectionFactory`, which the server has created, creates a `TopicConnection`. A `TopicConnection` creates a `TopicSession`. A `TopicSession` creates a `TopicPublisher` or `TopicSubscriber`.

• A topic subscriber (or queue receiver) can filter messages with a message selector. When a client specifies a message selector, the server will only send messages satisfying that filter to the client. The message selector syntax is based on SQL92.

• Message-driven EJBs, or message-driven beans, are a new type of Enterprise JavaBean available in Enterprise JavaBeans version 2.0, which is part of the Java 2 Enterprise Edition version 1.3.

• Message-driven beans are capable of processing JMS messages posted to a queue or topic.

• When a message is received, the EJB container uses any available instance of a particular message-driven bean to process the message.

• Since any message-driven EJB instance may be used, message-driven beans are not specific to particular client and must not maintain client state information.

• Any given EJB instance may process messages from multiple clients.

• Unlike session and entity beans (which require interfaces), message-driven beans require only that developers to provide the bean implementation class.

• Interface `MessageDrivenBean` declares `setMessageDrivenContext` and specifies that the message-driven context should be stored as instance variable `messageDrivenContext`.

**TERMINOLOGY**

- **BytesMessage** interface
- durable subscription
- Java Message Service (JMS)
- **MapMessage** interface
- message
- message body
- message consumer
- message-driven bean
- message header
- message-oriented middleware (MOM)
- message property
- message selector
- messaging system
- nondurable subscription
- **ObjectMessage** interface
- point-to-point messaging model
- publish
- publish/subscribe messaging model
- publisher

- **Queue**
- **QueueConnection** interface
- **QueueConnectionFactory** interface
- **QueueReceiver** interface
- **QueueSender** interface
- **QueueSession** interface
- receiver
- sender
- **StreamMessage** interface
- subscribe
- subscriber
- subscription
- **TextMessage** interface
- topic
- **TopicConnection** interface
- **TopicConnectionFactory** interface
- **TopicPublisher** interface
- **TopicSession** interface
- **TopicSubscriber** interface

**SELF-REVIEW EXERCISES**

16.1 State which of the following are true and which are false. If false, explain why.

a) Messages in the point-to-point messaging model are intended for zero or more recipients.

b) Messages in the publish/subscribe messaging model are intended for zero or one recipient.
c) When a message selector is specified, filtering is done on the server side.
d) The server maintains messages published to a topic while there are no subscriptions until a subscription is created.
e) The server maintains messages sent to a queue while there is no receiver until a receiver connects.
f) Message-driven beans maintain state for a specific client.

16.2 Fill in the blanks in each of the following:
a) The two messaging models are __________ and __________.
b) In the __________ messaging model, the client sends a message a __________ and intends that message for exactly one recipient.
c) In the __________ messaging model, the client sends a message to a __________ and intends that message for zero or more recipients.
d) The server will maintain messages for a __________ subscription while the subscription is inactive.
e) A __________ bean is a type of enterprise bean, which integrates well with MOM.

ANSWERS TO SELF-REVIEW EXERCISES

16.1 a) False. Messages in the point-to-point messaging model are intended for exactly one recipient.
b) False. Messages in the publish/subscribe messaging model are intended for zero or more recipients.
c) True. d) False. If there are no current subscriptions, the server will not maintain an incoming message; note that if there is a durable subscription that is inactive, however, the server will maintain messages for that subscription.
e) True. f) False. A message-driven bean instance may process messages from multiple clients; message-driven beans cannot maintain client-specific state.

16.2 a) Point-to-point, Publish/subscribe. b) Point-to-point, Queue. c) Publish/subscribe, Topic.
d) Durable. e) Message-driven.

EXERCISES

16.3 What is the purpose of a messaging system?

16.4 Compare and contrast the point-to-point messaging model and the publish/subscribe messaging model. When would it be appropriate to use one over the other?

16.5 Using the point-to-point messaging model, create an application that allows a seller to receive bids on an item. The offering message should include the bidder’s email address and bid price. (Hint: The seller should be the receiver of the offering messages, and the bidders should be the senders of the offering messages.)

16.6 Modify your solution to Exercise 16.5 to allow the seller to filter out offers lower than a certain bid price. (Hint: Set the bid price as a double property of the offering messages and use this property in a message selector.)

16.7 Create an application using the publish/subscribe messaging model that accepts orders from a client. The order should be published as a message to a Domestic_orders topic if the client’s shipping address is domestic or an International_orders topic if the client’s shipping address is international.

16.8 Modify your solution to Exercise 16.7 to allow subscribers of the Domestic_orders and International_orders topics to filter out product orders by category. For instance, suppose a component is responsible for book orders—allow that component to filter out all orders not of the "book" category. (Hint: Use a String property to set the order category and a message selector to filter by that property.)
Enterprise Java Case Study: Architectural Overview

Objectives

- To understand the Deitel Bookstore Enterprise Java case study architecture.
- To understand the decisions made in designing the Deitel Bookstore.
- To understand the Model-View-Controller (MVC) architecture in an Enterprise Java application context.
- To understand how XML and XSLT can generate content for many different client types.
- To understand the roles that servlets and EJBs play in enterprise applications.
- To understand multi-tier application design within the J2EE framework.

*It is easier to go down a hill than up, but the view is from the top.*
Arnold Bennett

*The universe is wider than our views of it.*
Henry David Thoreau

*Don’t bother looking at the view—I have already composed it.*
Gustav Mahler

*A whole is that which has a beginning, middle and end.*
Aristotle

*The life of the law has not been logic: it has been experience.*
Oliver Wendell Holmes, Jr.
17.1 Introduction

The technologies that comprise the Java 2 Enterprise Edition (J2EE) enable developers to build robust, scalable enterprise applications. In this case study, we build an online bookstore e-business application using several features of J2EE, including servlets and Enterprise JavaBeans, and other technologies, such as XML, XSLT, XHTML, WML and cHTML.

We used the Model-View-Controller (MVC) architecture in Chapter 5 to build a substantial drawing application. The Deitel Drawing application used MVC to separate the underlying representation of a drawing (a Collection of MyShapes) from views of that drawing (rendered using Java 2D) and input-processing logic (MyShapeControllers and a DragAndDropController). In this case study, we employ the MVC architecture to separate data and business logic from presentation logic and controller logic in a Java 2 Enterprise Edition application. A relational database and entity EJBs comprise the applications model. Java servlets implement controller logic for processing user input, and XSL transformations implement the application’s presentation logic.

XSLT presentation logic enables our application to present content to several client types. XSL transformations process application data (marked up with XML) to generate XHTML, WML and other presentations dynamically. We can extend the application to support additional client types and customize output for certain client types by implementing additional XSL transformations. For example, we could develop a J2ME MIDlet for handheld devices and implement a set of XSL transformations that produce output suitable for that MIDlet.

In this chapter, we present an overview of the Deitel Bookstore case study architecture. In the following chapters, we present the controller logic implementation with servlets (Chapter 18) and the business logic and data abstraction implementations with EJBs (Chapters 19 and 20). In Chapter 20, we also provide instructions for deploying the Deitel Bookstore case study on Sun Microsystems’ J2EE reference implementation application server. In Chapter 21, Application Servers, we introduce three of the top J2EE-compliant commercial application servers—BEA’s WebLogic, IBM’s WebSphere and the iPlanet Application Server. We discuss the features of each application server, then deploy the Deitel Bookstore case study on BEA’s WebLogic and IBM’s WebSphere.
17.2 Deitel Bookstore

The application we develop in this case study implements a subset of the functionality that customers expect of commercial on-line stores. We provide a product catalog that customers can search or browse for available books. We also provide a shopping cart to which customers can add products to be purchased. The customer can view the shopping cart, remove products, change the quantity of any product or purchase the products.

We provide customer registration, in which customers enter billing and shipping information. We also allow customers to view the details of previous orders and recover lost passwords. Customers can access the on-line store, using standard Web browsers, Wireless Markup Language (WML) browsers and cHTML (i-mode) browsers.

17.3 System Architecture

The Deitel Bookstore is a multi-tier application. Multi-tier applications—sometimes referred to as n-tier applications—divide functionality into separate tiers. Each tier may be located on a separate physical computer. We use a three-tier architecture in the Deitel Bookstore. Figure 17.1 presents the basic structure of three-tier applications.

The information tier (also called the data tier or the bottom tier) maintains data for the application. The information tier of an enterprise application typically stores data in a relational database management system (RDBMS). In the Deitel Bookstore case study, the database contains product information, such as a description, price and quantity in stock, and customer information, such as a user name, billing address and credit-card number.

The middle tier implements business logic and controller logic that control interactions between application clients and application data. The middle tier acts as an intermediary between the data in the information tier and the application clients. The middle-tier controller logic processes client requests (e.g., a request to view the product catalog) and retrieves data from the database. The middle-tier presentation logic then processes data from the information tier and presents the content to the client.

Software Engineering Observation 17.1

The Web server in a multi-tier application could be considered to be a separate tier, resulting in a four-tier application. We consider the Web server to be part of the middle tier in this case study, because the Web server simply delegates requests to the application server and forwards responses to the client.
Business logic enforces *business rules* and ensures that data are reliable before updating the database or presenting data to the user. Business rules dictate how clients of the application can and cannot access data, and how data are processed within the application. For example, an on-line store could have a business rule requiring that a customer’s credit-card issuer verify the customer’s credit card before the warehouse can ship the customer’s order. Business logic might implement this business rule by obtaining the credit-card number, expiration date and billing address from the customer and performing the verification. If the verification is successful, the business logic would update the database to indicate that the warehouse can ship the customer’s order.

The middle tier also implements the application’s presentation logic. Web applications typically present information to clients as XHTML documents. With recent advances in wireless technologies, many Web applications also present information to wireless clients as WML and cHTML documents. The middle tier of the Deitel Bookstore uses XML and XSLT to generate content for different client types dynamically, enabling support for Web browsers (XHTML), WAP browsers (WML) and i-mode browsers (cHTML).

The *client tier*, or top tier, is the application’s user interface. For Web applications, the client tier typically consists of a Web browser. Users view application output in the Web browser and click hyperlinks and form buttons to interact with the application. The Web browser then communicates with the middle tier to make requests and retrieve data from the information tier. The Deitel Bookstore supports Web, WML and cHTML browsers in the client tier. Developers can add support for other clients by providing XSL transformations for those other clients. Figure 17.2 presents a detailed diagram of the Deitel Bookstore enterprise application architecture. We discuss each portion of the diagram in the following sections.

### 17.4 Enterprise JavaBeans

Enterprise JavaBeans (EJBs) implement the Deitel Bookstore’s business logic and database abstraction layer. The primary business logic component is a stateful session EJB that represents a customer’s shopping cart. The entity EJBs in the Deitel Bookstore, which provide an object-based interface to the information tier, implement the Deitel Bookstore application’s model. Any program that can communicate using RMI-IIOP can use the EJB business logic. For example, an administrative tool could be developed as a stand-alone Java application that uses EJB business logic to modify application data. Servlets use the EJB business logic in the Deitel Bookstore application to create an on-line store.

#### 17.4.1 Entity EJBs

Entity EJBs provide an object-based abstraction of the application’s information tier. Each entity EJB represents a particular object stored in the application’s relational database. Instances of each entity EJB represent individual rows in the database. For example, a *Customer* EJB instance represents a store customer. The database stores each customer’s first name, last name, billing address, shipping address and credit-card information. Each *Customer* EJB instance represents a particular customer and provides methods for retrieving and storing the customer’s information.
To facilitate transmitting data through the application, each entity EJB has a corresponding model class that has properties for each entity EJB property. For example, the Product EJB, which represents a product in the database, has a corresponding ProductModel class with properties for the product’s ISBN, price, author, etc. Each model class implements interface Serializable and therefore is suitable for transmission over RMI-IIOP. Encapsulating data into model classes relieves network congestions by reducing the number of remote method calls required to obtain information from an entity EJB. For example, a servlet can invoke method getProductModel to obtain informa-
tion about a Product, instead of invoking separate methods such as getISBN, get-Price, getAuthor, etc. Each model class also implements interface XMLGenerator, which declares method getXML for retrieving an XML representation of a particular model class instance. The servlets in the Deitel Bookstore use these XML elements to build XML documents, such as the product catalog and order history.

17.4.2 Stateful Session EJBs

The ShoppingCart stateful session EJB, which manages the customer’s shopping cart, is the primary business-logic component in the Deitel Bookstore. Sometimes, customers browse online stores and add products to their shopping carts, then decide not to purchase those products. Such shopping carts are said to be abandoned. Rather than store an abandoned shopping cart in the database, the Deitel Bookstore uses a stateful session EJB to parallel more closely a user’s experience in brick-and-mortar stores. When a customer abandons a shopping cart, the EJB container removes the ShoppingCart EJB instance.

17.5 Servlet Controller Logic

Servlets provide the middle-tier interface between the client and the EJB business logic. The servlets in the Deitel Bookstore implement the controller in the application’s MVC architecture. Servlets handle client requests (via HTTP and WAP) and interact with the EJB business-logic components to fulfill those requests. The servlets then process data retrieved from the EJBs and generate XML documents that represent those data. These XML documents act as intermediate models of application data. The servlets then pass those XML documents through XSL transformations, which produce presentations for each client type.

17.6 XSLT Presentation Logic

Each servlet in the Deitel Bookstore employs an XSL Transformer and XSL transformations to generate appropriate presentations for each client type. The application requires a separate set of XSL transformations for each supported client type. For example, we provide one set of XSL transformations for producing XHTML, a second set for producing WML and a third set for producing cHTML. The servlets use a configuration file to determine the appropriate XSL transformation to apply for a particular client type.

GetProductServlet obtains an XML description of a product from that product’s ProductModel. An XSL Transformer uses an XSL transformation to extract data from the XML document and create a presentation for the client. If the client is a Web browser, the XSL Transformer uses an XSL transformation that produces XHTML. If the client is a WAP browser (e.g., running on a cell phone), the XSL Transformer uses an XSL transformation that produces WML.

Figure 17.3 shows a sample XML document generated by GetProductServlet. This XML document marks up a product, including the product’s ISBN, title, author, publisher, price, etc.

The XSL document of Fig. 17.4 transforms GetProductServlet’s XML document into XHTML, which is rendered in a Web browser in Fig. 17.5. The transformation simply extracts the relevant pieces of information from the XML document and creates an appropriate XHTML representation. We discuss the structures of these XSL transformations in Chapter 18.
<?xml version="1.0" encoding="UTF-8"?><!-- ProductDetails.xsl                                    --><!-- XSLT stylesheet for transforming content generated by --><!-- GetProductServlet into XHTML.                         --><xsl:stylesheet version = "1.0"xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">  <xsl:output method = "xml" omit-xml-declaration = "no" indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
<!-- include template for processing error elements --><xsl:include href = "/XSLT/XHTML/error.xsl"/>
<!-- template for product element --><xsl:template match = "product">  <html xmlns = "http://www.w3.org/1999/xhtml"xml:lang = "en" lang = "en">
  <head>
    <title>
      <xsl:value-of select = "title"/> -- Description
    </title>
    <link rel = "StyleSheet" href = "styles/default.css"/>
  </head>
  <body>
    <!-- copy navigation header into XHTML document -->
    <xsl:for-each select = "document( '/XSLT/XHTML/navigation.xml' )">
      <head>
        <title>
          <xsl:value-of select = "title"/> -- Description
        </title>
        <link rel = "StyleSheet" href = "styles/default.css"/>
      </head>
    </xsl:for-each>
  </body>
</xsl:template></xsl:stylesheet>
Fig. 17.4 XSL transformation for generating XHTML from GetProductServlet (part 2 of 3).
Fig. 17.4  XSL transformation for generating XHTML from GetProductServlet (part 3 of 3).

Fig. 17.5  XHTML document generated by XSLT in GetProductServlet (part 1 of 3).
Fig. 17.5  XHTML document generated by XSLT in GetProductServlet (part 2 of 3).
The XSL transformation of Fig. 17.6 transforms GetProductServlet’s XML document into WML, which appears in a WML browser in Fig. 17.7. Note that in the WML document there is little formatting information. WML is rendered on small devices, such as cell phones, so display capabilities are limited. These devices also have limited network connections, so the amount of data sent should be kept to a minimum. For example, the WML document does not contain an image of the book’s cover, which would be cumbersome to download over a wireless connection.

```
<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:output method = "xml" omit-xml-declaration = "no"
doctype-system = "http://www.wapforum.org/DTD/wml_1.1.xml"
doctype-public = "-//WAPFORUM//DTD WML 1.1//EN"/>
<xsl:include href = "/XSLT/WML/error.xsl"/>
```
Chapter 17  Enterprise Java Case Study: Architectural Overview

<xsl:template match = "product">
  <wml>
    <card id = "product" title = "{title}"
      <do type = "accept" label = "Add To Cart">
        <go href = "AddToCart" method = "post">
          <postfield name = "ISBN" value = "{ISBN}"/>
        </go>
      </do>
      <do type = "prev" label = "Back">
        <prev/>
      </do>
      <p>Description: </p>
      <p><xsl:value-of select = "title"/></p>
      <p>by <xsl:value-of select = "author"/></p>
      <p>
        <table columns = "2" title = "info">
          <tr>
            <td>ISBN: </td>
            <td><xsl:value-of select = "ISBN"/></td>
          </tr>
          <tr>
            <td>Price: </td>
            <td>${<xsl:value-of select = "price"/>}</td>
          </tr>
          <tr>
            <td>Publisher: </td>
            <td><xsl:value-of select = "publisher"/></td>
          </tr>
          <tr>
            <td>Pages: </td>
            <td><xsl:value-of select = "pages"/></td>
          </tr>
        </table>
      </p>
    </card>
  </wml>
</xsl:template>

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
   "http://www.wapforum.org/DTD/wml_1.1.xml"/>

Fig. 17.6  XSL transformation for generating WML from GetProductServlet
(part 2 of 2).
<wml>
  <card title="XML How to Program" id="product">
    <do label="Add To Cart" type="accept">
      <go method="post" href="AddToCart">
        <postfield value="0130284173" name="ISBN"/></go>
    </do>
    <do label="Back" type="prev"><prev/></do>
    <p>Description:</p>
    <p>XML How to Program</p>
    <p>by Deitel, Deitel, Nieto, Lin &amp; Sadhu</p>
    <table title="info" columns="2">
      <tr>
        <td>ISBN: 0130284173</td>
        <td>Price: $$69.95</td>
      </tr>
      <tr>
        <td>Publisher: Prentice Hall</td>
        <td>Pages: 1100</td>
      </tr>
    </table>
  </card>
</wml>

Fig. 17.7  WML document generated by XSLT in GetProductServlet (part 2 of 2). (Image © 2001 Nokia Mobile Phones.)
The XSL document of Fig. 17.8 transforms `GetProductServlet`’s XML document into cHTML, which appears in a cHTML browser in Fig. 17.9.

```xml
<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:output method = "html"
omit-xml-declaration = "yes"
indent = "yes"
doctype-system = "http://www.w3.org/MarkUp/html-spec/html-spec_toc.html"
doctype-public = "-//W3C//DTD HTML 2.0//EN"/>
<xsl:include href = "/XSLT/cHTML/error.xsl"/>
<xsl:template match = "product">
<html>
<head>
<title>
<xsl:value-of select = "title"/> -- Description
</title>
</head>
<body>
<div class = "header">
<xsl:value-of select = "title"/>
</div>
<div class = "author">
by <xsl:value-of select = "author"/>
</div>
<!-- create div element with details of Product -->
<div class = "productDetails">
<table>
<tr>
<td style = "text-align: center;"/>
<img class = "bookCover"
src = "images/{image}"/>
</td>
<td>
<p style = "text-align: right;"
Price: <xsl:value-of select = "price"/>
</p>
<p style = "text-align: right;"
</p>
</td>
</tr>
</table>
</div>
</body>
</html>
```

Fig. 17.8 XSL transformation for generating cHTML from `GetProductServlet` (part 1 of 2).
Fig. 17.8  XSL transformation for generating cHTML from GetProductServlet (part 2 of 2).

```xml
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.0//EN"
"http://www.w3.org/MarkUp/html-spec/html-spec_toc.html">
<html>
<head>
  <META http-equiv="Content-Type"
    content="text/html; charset=UTF-8">
  <title>XML How to Program -- Description</title>
</head>
<body>
  <div class="header">XML How to Program</div>
  <div class="author">
    by Deitel, Deitel, Nieto, Lin &amp; Sadhu
  </div>
  <div class="productDetails">
    <table>
      <tr>
        <td style="text-align: center;">
          <img src="images/xmlhtp1.jpg" class="bookCover">
        </td>
      </tr>
    </table>
  </div>
</body>
</html>
```

Fig. 17.9  cHTML document generated by XSLT in GetProductServlet (part 1 of 2). (Image courtesy of Pixo, Inc.)
<td>

<p style="text-align: right;">Price: $69.95</p>

<p style="text-align: right;">ISBN: 0130284173</p>

<p style="text-align: right;">Pages: 1100</p>

<p style="text-align: right;">Publisher: Prentice Hall</p>

<form action="AddToCart" method="post">
    <input value="Add to cart" type="submit">
    <input value="0130284173" name="ISBN" type="hidden">
</form>

</td>

</tr>

</table>

</div>

</body>

</html>

Fig. 17.9  cHTML document generated by XSLT in GetProductServlet (part 2 of 2). (Image courtesy of Pixo, Inc.)
This chapter overviewed the Deitel Bookstore case study architecture, which uses powerful enterprise Java capabilities, including servlets, EJBs, RMI, XML and XSLT. Chapters 18, 19 and 20 present the implementations of each tier. In Chapters 20–21, we provide instructions for deploying the Deitel Bookstore using the J2EE reference implementation, BEA’s WebLogic and IBM’s WebSphere.

**SUMMARY**

- The MVC design pattern, as applied in this enterprise application case study, separates data and business logic from presentation logic and controller logic.

- Multi-tier applications—sometimes referred to as n-tier applications—are divided into modular parts called tiers. Each tier may be located on a separate physical computer.

- The information tier, or data tier, maintains data for the application. The information tier typically stores data in a relational database management system (RDBMS). The database could contain product information, such as a description, price and quantity in stock, and customer information, such as a user name, billing address and credit-card number.

- The middle tier implements business logic and presentation logic to control interactions between application clients and application data. The middle tier acts as an intermediary between the data in the information tier and application clients.

- Middle-tier controller logic processes client requests (e.g., a request to view the product catalog) and retrieves data from the database. The middle-tier presentation logic then processes data from the information tier and presents the content to the client.

- Business logic enforces business rules and ensures that data are reliable before updating the database or presenting data to the user. Business rules dictate how clients of the application can and cannot access data and how data are processed within the application.

- The middle tier also implements the application’s presentation logic. The middle tier accepts client requests, retrieves data from the information tier and presents content to clients.

- Web applications typically present information to clients as XHTML documents. Many Web applications present information to wireless clients as Wireless Markup Language (WML) documents or Compact HyperText Markup Language (cHTML) documents.

- The middle tier of the Deitel Bookstore uses XML and XSLT to generate content for different client types dynamically, enabling support for XHTML, WML, cHTML and virtually any other client type.

- The client tier is the application’s user interface. Users interact directly with the application through the user interface. The client interacts with the middle tier to make requests and retrieve data from the application. The client then displays data retrieved from the middle tier to the user.

- Enterprise JavaBeans (EJBs) implement the Deitel Bookstore’s business rules. The entity EJBs in the Deitel Bookstore, along with the database they represent, are the model for the Deitel Bookstore application.

- Any program that can communicate using RMI-IIOP can use EJB business logic and database abstraction objects. For example, an administrative tool could be developed as a stand-alone Java application that uses the EJB business logic to process orders.

- Entity EJBs provide an object-based representation of the application’s information tier. Each entity EJB represents a particular object stored in the database. Instances of each entity EJB represent individual rows in the database.

- On-line store customers often browse the store catalog and add products to their shopping carts, but decide not to purchase the products. The shopping cart is therefore abandoned. Rather than
store the abandoned shopping cart in the database, we use session EJBs to parallel more closely a user’s experience in brick-and-mortar stores.

- Servlets provide the middle-tier interface between the client and the EJB business logic. The servlets in the Deitel Bookstore implement the controller of the MVC architecture. Servlets handle user requests (via HTTP and WAP) and interact with the EJB business logic.

- Servlets process data retrieved from EJBs and generate XML representations, which act as intermediate models of application data.

- XSL transformations act as the application view by transforming the XML model to appropriate formats for various client types.

- Each servlet in the Deitel Bookstore employs an XSL Transformer and XSL transformations to generate appropriate content for each client type. A separate set of XSL transformations generates content for each client type.

- The servlets produce XML documents to which the XSL Transformer applies XSL transformations to generate the appropriate content. To support additional client types, the developer can create additional sets of XSL transformations.

**TERMINOLOGY**

- abandoned shopping cart
- application server
- bottom tier
- brick-and-mortar store
- business logic
- business rules
- business-logic component
- client tier
- Compact HyperText Markup Language (cHTML)
- container-managed persistence
- controller logic
- controller-logic component
- data tier
- delegate
- design patterns
- e-business
- e-business application
- e-commerce
- e-commerce application
- Enterprise JavaBean (EJB)
- entity EJB
- four-tier application
- information tier
- middle tier
- model-view-controller (MVC)
- multi-tier application
- n-tier application
- presentation logic
- shopping cart e-commerce model
- three-tier architecture
- top tier
- Wireless Access Protocol (WAP)
- Wireless Markup Language (WML)
- XHTML
- XML
- XSL transformation
- XSL Transformer
- XSLT

**SELF-REVIEW EXERCISES**

17.1 What architectural pattern plays a major role in the Deitel Bookstore application architecture? What benefits does this design pattern provide?

17.2 Which part of the Deitel Bookstore application architecture could be separated into an additional tier? Why?

17.3 Section 17.3 describes a hypothetical business rule requiring credit-card verification. Describe an additional business rule appropriate for the Deitel Bookstore.

17.4 What are the controller-logic components in the Deitel Bookstore? What are the business-logic components? How do the controller-logic components in the application server communicate with the business-logic components?
17.5 What type of EJB in the Deitel Bookstore closely parallels customer experiences in brick-and-mortar stores?

17.6 How is the Deitel Bookstore application able to present content to clients of virtually any type?

17.7 How could a B2B partner (e.g., a corporate client) communicate with the Deitel Bookstore application? For example, a B2B partner may wish to order, on a periodic basis, large numbers of copies of a particular book used for training new employees. How could this B2B partner order the books programmatically?

ANSWERS TO SELF-REVIEW EXERCISES

17.1 The Deitel Bookstore application architecture uses the Model-View-Controller (MVC) architecture. MVC separates business logic from controller logic and presentation logic. Implementations of the business logic, controller logic and presentation logic can change independently of one another without requiring changes in the other components.

17.2 The Web server could be separated into a fourth tier. Separating the Web server into a fourth tier would provide a more modular design for applications that support clients using other protocols, in addition to HTTP.

17.3 An additional business rule could require customers to provide their e-mail addresses in the registration. Business logic could process the registration forms and reject registrations that do not include e-mail addresses.

17.4 Servlets are the controller-logic components in the Deitel Bookstore. Enterprise JavaBeans are the business-logic components. The servlets communicate with the EJBs through the EJB container using RMI-IIOP.

17.5 The ShoppingCart EJB is implemented as a stateful session EJB to simulate a shopping cart in a brick-and-mortar store.

17.6 The Deitel Bookstore application uses XSL transformations to present content for each client type. The servlets generate XML documents, and the developer supplies XSL transformations to transform the servlet XML documents for each client type.
Objectives

- To understand the role of presentation logic in a multitier application.
- To understand the role of controller logic in a multitier application.
- To understand the use of servlets in the middle tier of an Enterprise Java application.
- To understand how XML and XSLT can enable support for a variety of client types in a Web application.
- To understand how servlet initialization parameters can make servlets more flexible.

First come, first served.
Traditional Proverb

’Tis distance lends enchantment to the view...
Thomas Campbell

Invention is nothing more than a fine deviation from, or enlargement on a fine model...
Edward G. Bulwer-Lytton

Do not become the slave of your model.
Vincent Van Gogh

While Genius was thus wasting his strength in eccentric flights, I saw a person of a very different appearance, named Application.
Anna Letitia Barbauld
18.1 Introduction

In this chapter, we present the controller and presentation logic for the Deitel Bookstore. Controller logic in an application is responsible for processing user requests. For example, when a customer makes a request to add a book to a shopping cart, controller logic handles the request and invokes the appropriate business-logic methods to perform the requested action. The presentation logic then shows the output from the request to the user.

Java servlets in the implement the Deitel Bookstore’s controller logic. After invoking business-logic methods to process client requests, the servlets generate XML documents that contain content to presented to the client. These XML documents are not specific to any particular type of client (e.g., Web browser, cell phone, etc.); they simply mark up the data that the business logic supplies. XSL transformations implement the application’s presentation logic—the view in the MVC architecture—by transforming the XML documents into output suitable for each client type. For example, an XSL transformation might generate an XHTML document to present to a Web browser on a desktop computer, or a WML document to present to a WAP browser in a cell phone.

Using XSL transformations as presentation logic enable developers to extend the set of client types that the application supports without modifying the controller logic or business logic. To support a new client type, the developer simply supplies an additional set of
XSL transformations that produce appropriate output for the new client type. The developer then modifies an XML configuration file to install support for the new client type. [Note: The Deitel Bookstore requires the Apache Software Foundation’s Xalan XSL transformer. Download Xalan version 2.1 from xml.apache.org/dist/xalan-j and copy xalan.jar and xerces.jar to your J2SE SDK’s jre/lib/ext directory.]

This chapter is the first of three chapters that present the implementation of the Deitel Bookstore case study. In Chapters 19 and 20 we present the business logic and entity EJB components required for processing orders and working with the database.

### 18.2 XMLServlet Base Class

Each servlet in the Deitel Bookstore case study extends class XMLServlet (Fig. 18.1), which provides common initialization and utility methods. To enable support for multiple client types, each servlet in the Deitel Bookstore creates an XML document that represents raw application data retrieved from the application’s business logic and information tier. An XSLT transformation processes this XML document and presents the output to the client. Class XMLServlet implements this common functionality.

Method init (lines 41–85) initializes the servlet. The servlet container provides the ServletConfig argument when the container initializes the servlet. Line 49 retrieves the name of the XSL transformation that will transform the servlet’s content for each client type. This file name is provided as a servlet-initialization parameter, so the deployer must define this parameter when deploying the servlet. Line 52 creates a DocumentBuilderFactory instance, which will be used to create DocumentBuilders for building XML documents. Line 55 creates a TransformerFactory, with which instances of XMLServlet perform XSL transformations. Lines 58–81 set a URIResolver for the TransformerFactory to enable its Transformers to resolve relative URIs in XSL documents. For example, if an XSL document references another XSL document using element xsl:include with a relative URI (e.g., /XSLT/XHTML/error.xsl), the Transformer invokes method resolve of interface URIResolver to determine from where the Transformer should load the included XSL. Line 67 resolves the relative URI by invoking ServletContext method getResource, which returns a URL. Line 71 returns a StreamSource for the URL’s InputStream. The Transformer uses this StreamSource to read the contents of the included XSL document.

```java
package com.deitel.advjhttp1.bookstore.servlets;

import java.io.*;
import java.util.*;
import java.net.URL;
import javax.servlet.*;
import javax.servlet.http.*;
```

Fig. 18.1 XMLServlet base class for servlets in the Deitel Bookstore (part 1 of 8).
```java
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// third-party packages
import org.w3c.dom.*;
import org.xml.sax.SAXException;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;

public class XMLServlet extends HttpServlet {

    // factory for creating DocumentBuilders
    private DocumentBuilderFactory builderFactory;

    // factory for creating Transformers
    private TransformerFactory transformerFactory;

    // XSL file that presents servlet's content
    private String XSLFileName;

    // ClientModel List for determining client type
    private List clientList;

    // initialize servlet
    public void init( ServletConfig config )
    throws ServletException
    {
        // call superclass's init method for initialization
        super.init( config );

        // use InitParameter to set XSL file for transforming
        // generated content
        setXSLFileName( config.getInitParameter( "XSL_FILE" ) );

        // create new DocumentBuilderFactory
        builderFactory = DocumentBuilderFactory.newInstance();

        // create new TransformerFactory
        transformerFactory = TransformerFactory.newInstance();

        // set URIResolver for resolving relative paths in XSLT
        transformerFactory.setURIResolver(new URIResolver() {
            // resolve href as relative to ServletContext
            public Source resolve( String href, String base )
            {
                try {
                    ServletContext context = getServletContext();
                    // ...
URL url = context.getResource( href );

// create StreamSource to read document from URL
return new StreamSource( url.openStream() );

// handle exceptions obtaining referenced document
catch ( Exception exception ) {
    exception.printStackTrace();

    return null;
}

// create ClientModel ArrayList
clientList = buildClientList();

// get DocumentBuilder instance for building XML documents
public DocumentBuilder getDocumentBuilder( boolean validating )
{
    // create new DocumentBuilder
    try {

        // set validation mode
        builderFactory.setValidating( validating );

        // return new DocumentBuilder to the caller
        return builderFactory.newDocumentBuilder();
    }

    // handle exception when creating DocumentBuilder
    catch ( ParserConfigurationException parserException ) {
        parserException.printStackTrace();

        return null;
    }

} // end method getDocumentBuilder

// get non-validating parser
public DocumentBuilder getDocumentBuilder()
{
    return getDocumentBuilder( false );
}

// set XSL file name for transforming servlet's content
public void setXSLFileName( String fileName )
{
    XSLFileName = fileName;

} // end setXSLFileName

Fig. 18.1  **XMLServlet** base class for servlets in the Deitel Bookstore (part 3 of 8).
// get XSL file name for transforming servlet's content
public String getXSLFileName()
{
    return XSLFileName;
}

// write XML document to client using provided response
// Object after transforming XML document with
// client-specific XSLT document
public void writeXML( HttpServletRequest request,
                     HttpServletResponse response, Document document )
    throws IOException
{
// get current session, create if not extant
    HttpSession session = request.getSession( true );

// get ClientModel from session Object
    ClientModel client = ( ClientModel )
        session.getAttribute( "clientModel" );

// if client is null, get new ClientModel for this
// User-Agent and store in session
    if ( client == null ) {
        String userAgent = request.getHeader( "User-Agent" );
        client = getClientModel( userAgent );
        session.setAttribute( "clientModel", client );
    }

// set appropriate Content-Type for client
    response.setContentType( client.getContentType() );

// get PrintWriter for writing data to client
    PrintWriter output = response.getWriter();

// build file name for XSLT document
    String xslFile = client.getXSLPath() + getXSLFileName();

// open InputStream for XSL document
    InputStream xslStream =
        getServletContext().getResourceAsStream( xslFile );

// transform XML document using XSLT
    transform( document, xslStream, output );

// flush and close PrintWriter
    output.close();
} // end method writeXML

// transform XML document using provided XSLT InputStream
// and write resulting document to provided PrintWriter

Fig. 18.1  XMLServlet  base class for servlets in the Deitel Bookstore (part 4 of 8).
public void transform( Document document,
    InputStream xslStream, PrintWriter output )
{
    // create Transformer and apply XSL transformation
    try {
        // create DOMSource for source XML document
        Source xmlSource = new DOMSource( document );

        // create StreamSource for XSLT document
        Source xslSource =
            new StreamSource( xslStream );

        // create StreamResult for transformation result
        Result result = new StreamResult( output );

        // create Transformer for XSL transformation
        Transformer transformer =
            transformerFactory.newTransformer( xslSource );

        // transform and deliver content to client
        transformer.transform( xmlSource, result );
    } // end try

    // handle exception when transforming XML document
    catch ( TransformerException transformerException ) {
        transformerException.printStackTrace();
    } // end catch

} // end method transform

// build error element containing error message
public Node buildErrorMessage( Document document,
    String message)
{
    // create error element
    Element error = document.createElement( "error" );

    // create message element
    Element errorMessage =
        document.createElement( "message" );

    // create message text and append to message element
    errorMessage.appendChild( document.createTextNode( message ) );

    // append message element to error element
    error.appendChild( errorMessage );

    return error;
}
private List buildClientList()
{
    DocumentBuilder builder = getDocumentBuilder( true );

    List clientList = new ArrayList();

    String clientXML = getServletContext().getInitParameter("CLIENT_LIST");

    try {
        InputStream clientXMLStream =
            getServletContext().getResourceAsStream(clientXML);

        Document clientsDocument =
            builder.parse( clientXMLStream );

        NodeList clientElements =
            clientsDocument.getElementsByTagName("client");

        int listLength = clientElements.getLength();

        for ( int i = 0; i < listLength; i++ ) {
            Element client =
                ( Element ) clientElements.item( i );

            Element agentElement =
                ( Element )
                client.getElementsByTagName("userAgent").item( 0 );

            Text agentText =
                ( Text )
                agentElement.getFirstChild();

            String agent = agentText.getNodeValue();
        }
    }
}
// get contentType Element
Element typeElement = (Element) client.getElementsByTagName("contentType").item(0);

// get contentType Element's child text node
Text typeText = (Text) typeElement.getFirstChild();

// get value of contentType text node
String type = typeText.getNodeValue();

// get XSLPath element
Element pathElement = (Element) client.getElementsByTagName("XSLPath").item(0);

// get Text node child of XSLPath
Text pathText = (Text) pathElement.getFirstChild();

// get value of XSLPath text node
String path = pathText.getNodeValue();

// add new ClientModel with userAgent, contentType and XSLPath for this client Element
clientList.add(new ClientModel(agent, type, path));

} // end try

// handle SAXException when parsing XML document
catch (SAXException saxException) {
saxException.printStackTrace();
}

// catch IO exception when reading XML document
catch (IOException ioException) {
ioException.printStackTrace();
}

// return newly creating list of ClientModels
return clientList;

} // end method buildClientList

// get ClientModel for given User-Agent HTTP header
private ClientModel getClientModel(String header) {

    // get Iterator for clientList
    Iterator iterator = clientList.iterator();
    
    // get Iterator for clientList
    Iterator iterator = clientList.iterator();

Fig. 18.1 XMLServlet base class for servlets in the Deitel Bookstore (part 7 of 8).
Line 84 invokes method `buildClientList`, which reads the `clients.xml` configuration file and produces a `List` of `ClientModel`s (Fig. 18.4). Instances of `XMLServlet` use these `ClientModel`s to determine which XSL transformation to apply for each client type.

Method `getDocumentBuilder` (lines 88–107) creates a `DocumentBuilder` object for parsing and creating XML documents. The `boolean` argument specifies whether method `getDocumentBuilder` should create a validating XML parser. Class `XMLServlet` stores a `DocumentBuilderFactory` in an instance variable (line 29), which prevents the need to create a new `DocumentBuilderFactory` each time a `DocumentBuilder` is needed. The overloaded `getDocumentBuilder` (lines 110–113) invokes method `getDocumentBuilder` with a `false` argument to create a non-validating XML parser.

Methods `setXSLFileName` and `getXSLFileName` (lines 116–125) are `set` and `get` methods for the `XSLFileName` property of class `XMLServlet`. The `XSLFileName` property specifies the name of the XSLT document that transforms the servlet’s content for a particular client type. The Deitel Bookstore application separates the XSL transformations for the various client types into separate directories. For example, the file `products.xsl` in directory `/XSLT/XHTML` produces XHTML output, whereas the version of `products.xsl` in directory `/XSLT/WML` produces WML output. The `XSLFileName` property specifies only the filename (e.g., `products.xsl`); each `ClientModel` specifies the appropriate directory for a particular client type.

Method `writeXML` (lines 130–168) determines which type of client is accessing the servlet and invokes method `transform` to perform an XSL transformation. Lines 138–139 attempt to obtain a `ClientModel` from the `HttpServletRequest`. If the `HttpServletRequest` does not contain a `ClientModel`, line 144 obtains the client’s `User-Agent` header, which uniquely identifies the client type. Line 145 invokes method `getClientModel` to get an appropriate `ClientModel` for the given client type. Line 146 places the `ClientModel` in the `HttpServletRequest` for later use. Line 150 obtains the content type for the client
from the ClientModel and configure the HttpServletResponse object. Line 156 constructs the complete relative path for the XSL transformation by concatenating the ClientModel’s XSLPath property with the servlet’s XSLFileName property. Lines 159–160 open an InputStream for the XSL transformation. Line 163 invokes method transform to perform the transformation and send the result to the client.

Method transform (lines 172–202) performs the given XSL transformation on the given XML document, and writes the result of the transformation to the given PrintWriter. Line 179 creates a DOMSource for the XML document. Lines 182–183 create a StreamSource for the XSL document. Line 159 creates a StreamResult for the PrintWriter to which the Transformer will write the results of the XSL transformation. Lines 189–190 create a Transformer by invoking TransformerFactory method newTransformer. Line 193 invokes Transformer method transform to perform the XSL transformation on the given Source object and writes the result to the given Result object.

Method buildErrorMessage (lines 205–223) is a utility method for constructing XML Element that contains an error message. Line 209 creates an error Element using the provided document. Lines 212–213 create a errorMessage Element that will contain the actual error message. Lines 189–190 create a Text node that contains the text of the error message and append this Text node to the errorMessage Element. Line 220 appends the errorMessage Element to the error Element, and line 223 returns the complete error Element.

Method buildClientList (lines 227–323) constructs a List of ClientModels by reading client information from an XML configuration file (Fig. 18.2). Lines 237–238 retrieve the name of the configuration file from an initialization parameter in the servlet’s ServletContext. The deployer must specify the value of this parameter when deploying the application. Lines 244–246 open an InputStream to the configuration file. Lines 249–50 parse the XML configuration file and build a Document object in memory. Lines 253–254 get a NodeList of client Elements from the document. Each client Element has a name Attribute and three child Elements—user-Agent, contentType and XSLPath—each of which corresponds to a ClientModel property. Lines 260–306 construct a ClientModel for each client Element in the XML configuration file and add those ClientModels to the List.

Method getClientModel (lines 326–347) returns a ClientModel that matches the given User-Agent header. The userAgent property of class ClientModel contains a substring of the User-Agent header that uniquely identifies a particular client type. For example, the User-Agent substring Mozilla/4.0 (compatible; MSIE 5 uniquely identifies the Microsoft Internet Explorer version 5 client. Lines 333–341 compare the userAgent property of each ClientModel with the given User-Agent header. If the userAgent property is a substring of the User-Agent header, the ClientModel is a match and line 340 returns it to the caller. Lines 344–345 construct a generic XHTML Cli-
entModel as a default if no others match the given User-Agent header.

Figure 18.2 presents a sample configuration file that enables application support for several popular client types. Figure 18.3 presents the DTD for this configuration file.

Class ClientModel (Fig. 18.4) represents a particular client type. Lines 18–23 define the ClientModel constructor. Lines 26–59 provide set and get methods for the userAgent, ContentType and XSLPath properties.
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE clients SYSTEM
 "http://www.deitel.com/advjhtp1/clients.dtd">

<!-- Client configuration file for the Deitel Bookstore -->

<clients>

<!-- Microsoft Internet Explorer version 5 client -->
<client name = "Microsoft Internet Explorer 5">
  <userAgent>Mozilla/4.0 (compatible; MSIE 5</userAgent>
  <contentType>text/html</contentType>
  <XSLPath>/XSLT/XHTML/</XSLPath>
</client>

<!-- Microsoft Internet Explorer version 6 client -->
<client name = "Microsoft Internet Explorer 6">
  <userAgent>Mozilla/4.0 (compatible; MSIE 6</userAgent>
  <contentType>text/html</contentType>
  <XSLPath>/XSLT/XHTML/</XSLPath>
</client>

<!-- Netscape version 4.7x client -->
<client name = "Netscape 4.7">
  <userAgent>Mozilla/4.7</userAgent>
  <contentType>text/html</contentType>
  <XSLPath>/XSLT/XHTML/</XSLPath>
</client>

<!-- Mozilla/Netscape version 6 client -->
<client name = "Mozilla/Netscape 6">
  <userAgent>Gecko</userAgent>
  <contentType>text/html</contentType>
  <XSLPath>/XSLT/XHTML/</XSLPath>
</client>

<!-- Phone.com WML browser client -->
<client name = "Openwave SDK Browser">
  <userAgent>UP.Browser/4</userAgent>
  <contentType>text/vnd.wap.wml</contentType>
  <XSLPath>/XSLT/WML/</XSLPath>
</client>

<!-- Nokia WML browser client -->
<client name = "Nokia WAP Toolkit Browser">
  <userAgent>Nokia-WAP</userAgent>
  <contentType>text/vnd.wap.wml</contentType>
  <XSLPath>/XSLT/WML/</XSLPath>
</client>

<!-- Pixo iMode browser client -->
<client name = "Pixo Browser">
</client>

</clients>

Fig. 18.2 Configuration file for enabling support for various client types (clients.xml) (part 1 of 2).
Fig. 18.2  Configuration file for enabling support for various client types (clients.xml) (part 2 of 2).

```xml
<!ELEMENT clients ( client+ )>
<!ELEMENT client ( userAgent, contentType, XSLPath )>
<!ATTLIST client name CDATA #REQUIRED>
<!ELEMENT userAgent ( #PCDATA )>
<!ELEMENT contentType ( #PCDATA )>
<!ELEMENT XSLPath ( #PCDATA )>
```

Fig. 18.3  DTD for clients.xml.

```java
// ClientModel.java
// ClientModel is a utility class for determining the proper
// Content-Type and path for XSL files for each type of client
// supported by the Bookstore application.
package com.deitel.advjhtp1.bookstore.model;

// Java core packages
import java.io.*;

public class ClientModel implements Serializable {
    // ClientModel properties
    private String userAgent;
    private String contentType;
    private String XSLPath;

    // ClientModel constructor for initializing data members
    public ClientModel( String agent, String type, String path ) {
        setUserAgent( agent );
        setContentAgent( type );
        setXSLPath( path );
    }
}
```

Fig. 18.4  ClientModel for representing supported clients (part 1 of 2).
18.3 Shopping Cart Servlets

Several e-commerce models have become common for shopping on the Web. These models include auction sites, bargain shoppers, bartering and name your own price. Our application uses the familiar shopping-cart model, in which a customer browses through the store and selects items for purchase. Each of these items is placed in a virtual shopping cart. When the customer is finished shopping, a checkout process gathers the person’s billing and shipping information to complete the transaction. Maintaining the shopping cart is part of the business logic implemented in the application’s EJBs.

Figure 18.5 shows the flow of client requests and responses for a new customer who orders multiple copies of a book using a Web browser. The customer enters the site at index.html. The customer then selects a link to view the product catalog, which is generated by GetAllProductsServlet. The customer then chooses a product from the
product catalog, and **GetProductServlet** shows the details of the product, including a picture of the cover, the price and the author(s). The customer adds the product to the shopping cart through **AddToCartServlet**, and **ViewCartServlet** shows the shopping cart contents. The customer updates the quantity to be purchased through **UpdateCartServlet**, and **ViewCartServlet** again shows the shopping cart contents to the customer. The customer then selects a link to register as a new customer and fills out the registration form in **register.html**. The customer submits the form to **RegisterServlet**, which then invokes **LoginServlet** to log the customer into the store. The customer then selects a link to invoke **CheckoutServlet**. **CheckoutServlet** places the order and forwards the customer to **ViewOrderServlet**, which shows the details of the customer’s order.

### 18.3.1 AddToCartServlet

As customers browse through our on-line store, they add products they wish to purchase to their shopping carts. Servlet **AddToCartServlet** (Fig. 18.6) handles each customer request to add a product to the shopping cart. Class **AddToCartServlet** extends class **XMLServlet**, which implements functionality common to all servlets in the Deitel Bookstore, such as initialization and XSL transformations.

Line 41 retrieves from the **HttpServletRequest** object the ISBN of the product the customer would like purchase. Lines 37–38 retrieve a reference to the customer’s **ShoppingCart** from the servlet’s **HttpSession** object. It is possible that the customer does not yet have a shopping cart, in which case the **ShoppingCart** reference will be **null**. If this is the case, line 59 creates a new **ShoppingCart** using the **ShoppingCart** home interface, and stores its reference in the servlet’s **HttpSession** object (line 62) for later use. The **ShoppingCart** EJB is a stateful session bean and will therefore maintain the contents of the customer’s **ShoppingCart** throughout the browsing session.

Once the servlet has a valid **ShoppingCart** reference, line 66 invokes the **ShoppingCart**’s **addProduct** business method with the product’s ISBN as an argument. Line 70 invokes method **sendRedirect** of class **HttpServletResponse** to redirect the client to **ViewCartServlet**, which displays a list of products in the shopping cart.

There are three checked exceptions that could be thrown from the **try** block on lines 44–72. Lines 75–86 **catch a NamingException** if the **ShoppingCart** EJB is not found in the JNDI directory. Lines 89–99 **catch a CreateException** if there is an error creating the customer’s shopping cart. Lines 102–110 **catch a ProductNotFoundException**, which is an application-specific exception that we present in detail in Chapter 19. A **ProductNotFoundException** indicates that a product with the given ISBN could not be found in the database.

Each of these exception handlers uses **XMLServlet** method **buildErrorMessage** to create an XML error message to present to the client. In each case, we provide a message to inform the user of the error, and print a stack trace of the exception to aid the developer in debugging the application. Invoking **XMLServlet** method **writeXML** (lines 85, 98 and 109) sends the content to the client.

### 18.3.2 ViewCartServlet

Once the customer has added an item to the shopping cart, **ViewCartServlet** (Fig. 18.7) displays the shopping cart’s contents. This servlet is a subclass of **XMLServlet**.
Fig. 18.5 Flow of client requests and data returned in the Deitel Bookstore for XHTML clients.

```java
// AddToCartServlet.java
// AddToCartServlet adds a Product to the Customer's ShoppingCart.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;
```

Fig. 18.6 AddToCartServlet for adding products to a shopping cart (part 1 of 3).
public class AddToCartServlet extends XMLServlet {

    // respond to HTTP post requests
    public void doPost( HttpServletRequest request,
            HttpServletResponse response )
        throws ServletException, IOException {
        Document document = getDocumentBuilder().newDocument();

        // get HttpSession Object for this user
        HttpSession session = request.getSession();

        // get Customer's ShoppingCart
        ShoppingCart shoppingCart =
        ( ShoppingCart ) session.getAttribute( "cart" );

        // get ISBN parameter from request Object
        String isbn = request.getParameter( "ISBN" );

        // get ShoppingCart and add Product to be purchased
        try {
            InitialContext context = new InitialContext();

            // create ShoppingCart if Customer does not have one
            if ( shoppingCart == null ) {
                Object object = context.lookup( "java:comp/env/ejb/ShoppingCart" );

                // cast Object reference to ShoppingCartHome
                ShoppingCartHome shoppingCartHome =
                ( ShoppingCartHome )
                PortableRemoteObject.narrow( object, ShoppingCartHome.class );

                // create ShoppingCart using ShoppingCartHome
                shoppingCart = shoppingCartHome.create();

                // store ShoppingCart in session
                session.setAttribute( "cart", shoppingCart );
            }

            // add Product to Customer's ShoppingCart
            shoppingCart.addProduct( isbn );

            // redirect Customer to ViewCartServlet to view
            // contents of ShoppingCart
            response.sendRedirect( "ViewCart" );
        }
    }
}

Fig. 18.6 AddToCartServlet for adding products to a shopping cart (part 2 of 3).
namingException.printStackTrace();

String error = "The ShoppingCart EJB was not " +
"found in the JNDI directory."

// append error message to XML document
document.appendChild( buildErrorMessage(
document, error ) );

writeXML( request, response, document );
}

// handle exception when creating ShoppingCart EJB
try ( CreateException createException ) {
createException.printStackTrace();

String error = "ShoppingCart could not be created";

// append error message to XML document
document.appendChild( buildErrorMessage(
document, error ) );

writeXML( request, response, document );
}

// handle exception when Product is not found
try ( ProductNotFoundException productException ) {
productException.printStackTrace();

// append error message to XML document
document.appendChild( buildErrorMessage(
document, productException.getMessage() ) );

writeXML( request, response, document );
}
}

} // end method doGet

/* Fig. 18.6 AddToCartServlet for adding products to a shopping cart
(part 3 of 3). */

/* Fig. 18.7 ViewCartServlet for viewing contents of shopping cart
(part 1 of 3). */
```java
// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.rmi.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;

public class ViewCartServlet extends XMLServlet {

    // respond to HTTP get requests
    public void doGet( HttpServletRequest request,
                        HttpServletResponse response )
            throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();
        HttpSession session = request.getSession();

        // get Customer's ShoppingCart from session
        ShoppingCart shoppingCart =
            ( ShoppingCart ) session.getAttribute( "cart" );

        // build XML document with contents of ShoppingCart
        if ( shoppingCart != null ) {
            // create cart element in XML document
            Element root = ( Element ) document.appendChild(
                document.createElement( "cart" ) );

            // get total cost of Products in ShoppingCart
            double total = shoppingCart.getTotal();

            // create NumberFormat for local currency
            NumberFormat priceFormatter =
                NumberFormat.getCurrencyInstance();

            // format total price for ShoppingCart and add it
            // as an attribute of element cart
            root.setAttribute( "total",
                priceFormatter.format( total ) );

            // get contents of ShoppingCart
            Iterator orderProducts =
                shoppingCart.getContents().iterator();

            // add an element for each Product in ShoppingCart
            // to XML document
            while ( orderProducts.hasNext() ) {
                OrderProductModel orderProductModel =
                orderProducts.next();
```

Fig. 18.7 ViewCartServlet for viewing contents of shopping cart (part 2 of 3).
A remote reference to the customer’s ShoppingCart is stored in the HttpSession object. Lines 34–35 retrieve this remote reference. After ensuring the ShoppingCart remote reference is not null, lines 41–42 begin building the XML document that describes the customer’s shopping cart by creating element cart. Line 45 obtains the total price for the items in the shopping cart by invoking ShoppingCart method getTotal. Lines 48–49 format the total using a NumberFormat and append the formatted total to the XML document.


If the ShoppingCart remote reference stored in the HttpSession object was null, lines 72–78 generate an error message. Line 81 invokes method writeXML to transforms the XML content using XSLT and present that content to the client.

The XSL transformation of Fig. 18.8 produces XHTML for ViewCartServlet. Lines 6–8 set the output parameters for the transformation. Line 10 includes templates from error.xsl for transforming error messages. Lines 23–27 load a navigation header from an external XML document. Note that we refer to the error.xsl and navigation.xml documents with relative URIs. Recall that class XMLServlet creates a custom URIResolver that enables the XSL transformer to resolve these relative URIs. Had we not provided a custom URIResolver, each XSL transformation would have to specify the complete URI for these documents, which would make the application less portable. This header contains links for viewing the product catalog, creating an online
account, viewing the shopping cart, etc. Lines 42–43 apply a template that populates a table with product information, including the title, author, ISBN price and quantity. The form on lines 51–54 enables the customer to check out from the online store to place the order. The \texttt{xsl:template} on lines 61–91 extracts each product’s information from the \texttt{ViewCartServlet}’s XML document and creates a table row. The form on lines 75–80 enables the user to change the quantity of the product in the shopping cart. The form on lines 84–88 enables the user to remove the product from the shopping cart.

The XSL transformation of Fig. 18.9 produces cHTML for \texttt{ViewCartServlet}. Lines 28–29 display the total cost of products in the cart. Lines 38–41 apply the \texttt{xsl:template} for \texttt{orderProduct} elements to produce an unordered list of product information. The form on lines 44–47 enables the customer to check out of the online store to place the order. The form on lines 64–69 enables the customer to update the quantity of a particular product in the shopping cart. The form on lines 72–76 enables the customer to remove a product from the shopping cart.

\begin{Verbatim}
<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
    <xsl:output method = "xml" omit-xml-declaration = "no"
        indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
        doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
    <xsl:include href = /XSLT/XHTML/error.xsl"/>
    <xsl:template match = "cart">
        <html xmlns = "http://www.w3.org/1999/xhtml"
            xml:lang = "en" lang = "en">
            <head>
                <title>Your Online Shopping Cart</title>
                <link rel = "StyleSheet" href = "styles/default.css"/>
            </head>
            <body>
                <xsl:for-each select =
                    "document( '/XSLT/XHTML/navigation.xml' )">
                    <xsl:copy-of select = "."/>
                </xsl:for-each>
                <div class = "header">Your Shopping Cart:</div>
                <table class = "cart">
                    <tr>
                        <th>Title</th>
                        <th>Author(s)</th>
                    </tr>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
\end{Verbatim}

\textbf{Fig. 18.8} \texttt{ViewCartServlet} XSL transformation for XHTML browsers (\texttt{XHTML/viewCart.xsl}) (part 1 of 3).
<th>ISBN</th>
<th>Price</th>
<th>Quantity</th>
</tr>

<xsl:apply-templates
select = "orderProduct"/>
</table>
<p>Your total:
<xsl:value-of select = "@total"/>
</p>
<form action = "Checkout" method = "post">
<input name = "submit" type = "submit"
value = "Check Out"/>
</form>
</p>
</body>
</html>

<xsl:template match = "orderProduct">
<tr>
<td>
><xsl:value-of select = "product/title"/></a>
</td>
<td><xsl:value-of select = "product/author"/></td>
<td><xsl:value-of select = "product/ISBN"/></td>
<td><xsl:value-of select = "product/price"/></td>
<td>
<form action = "UpdateCart" method = "post">
<input type = "text" size = "2"
name = "{product/ISBN}"
value = "{quantity}"/>
<input type = "submit" value = "Update"/>
</form>
</td>
<td align = "center">
<form action = "RemoveFromCart" method = "post">
<input name = "ISBN" type = "hidden"
value = "{product/ISBN}"/>
</form>
</td>
</tr>

<xsl:template match = "orderProduct"/>
</body>
</html>

Fig. 18.8  ViewCartServlet XSL transformation for XHTML browsers (XHTML/viewCart.xsl) (part 2 of 3).
Fig. 18.8 **ViewCartServlet** XSL transformation for XHTML browsers (**XHTML/viewCart.xsl**) (part 3 of 3).

```
<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
    <xsl:output method = "html"
        omit-xml-declaration = "yes"
        indent = "yes"
        doctype-system = "http://www.w3.org/MarkUp/html-spec/html-spec_toc.html"
        doctype-public = "-//W3C//DTD HTML 2.0//EN"/>
    <xsl:include href = "/XSLT/cHTML/error.xsl"/>
</xsl:stylesheet>
```

Fig. 18.9 **ViewCartServlet** XSL transformation for i-mode browsers (**cHTML/viewCart.xsl**) (part 1 of 3). (Image courtesy of Pixo, Inc.)
<xsl:template match = "cart">
  <html>
    <head>
      <title>Your Online Shopping Cart</title>
    </head>
    <body>
      <div class = "header">
        Your Shopping Cart:
      </div>
      <p>
        Your total:
        <xsl:value-of select = "@total"/>
      </p>
      <p>Title</p>
      <p>Author(s)</p>
      <p>ISBN</p>
      <p>Price</p>
      <p>Quantity</p>
      <ul>
        <xsl:apply-templates
          select = "orderProduct"/>
      </ul>
      <p>
        <form method = "post" action = "Checkout">
          <input type = "submit" name = "Checkout" value = "Checkout"/>
        </form>
      </p>
    </body>
  </html>
</xsl:template>

<xsl:template match = "orderProduct">
  <li>
      <xsl:value-of select = "product/title"/>
    </a>
    <br/>
    <p><xsl:value-of select = "product/author"/></p>
    <p><xsl:value-of select = "product/price"/></p>
    <p>
      <form method = "post" action = "UpdateCart">
        <input type = "text" size = "2"
          name = "{product/ISBN}"/>
      </form>
    </p>
  </li>
</xsl:template>

Fig. 18.9 ViewCartServlet XSL transformation for i-mode browsers (cHTML/viewCart.xsl) (part 2 of 3). (Image courtesy of Pixo, Inc.)
Chapter 18 Enterprise Java Case Study: Presentation and Controller Logic

The XSL transformation of Fig. 18.10 produces WML for ViewCartServlet. The do element on lines 21–23 enables the user to check out of the online store to place the order. Lines 25–49 mark up the list of products in the shopping cart. The card element on lines 55–64 displays information about a single product. The card element on lines 66–81 provides an interface for changing the quantity of a particular product in the shopping cart. [Note: To download the Nokia WAP Toolkit WML browser, please visit www.nokia.com/corporate/wap/downloads.html.]
18.3.3 RemoveFromCartServlet

Once a customer has added a product to the shopping cart, the customer may wish to remove that product. **RemoveFromCartServlet** (Fig. 18.11) processes requests to remove products from shopping carts. Line 34 retrieves from the HttpSession the ISBN of the product that the customer would like to remove from the shopping cart. Lines 39–40 obtain the ShoppingCart remote reference from the HttpSession object. After ensuring that the ShoppingCart reference is not null, line 44 invokes ShoppingCart method removeProduct, providing the ISBN as an argument. Line 48 redirects the client to ViewCartServlet to display the results of removing the product from the shopping cart. Method removeProduct throws a ProductNotFoundException if a product with the given ISBN is not found in the shopping cart. Lines 53–61 catch this exception and generate an error message to inform the user of the problem.

18.3.4 UpdateCartServlet

Products in a customer’s shopping cart each have an associated quantity. **UpdateCartServlet** (Fig. 18.12) enables customers to change the quantities of products in their shopping carts. Lines 37–38 retrieve the customer’s ShoppingCart from the HttpSession object. Line 44 retrieves an Enumeration of parameter names from the HttpServletRequest object. Each parameter name is the ISBN of a product in the customer’s shopping cart. The value of each parameter is the desired quantity of the product in the shopping cart. For each product in the shopping cart, lines 50–51 obtain the newQuantity for that product from the request parameter. Lines 55–56 set the quantity for each product in the shopping cart by invoking ShoppingCart method setProductQuantity. Once all of the quantities have been updated, line 61 redirects the client to ViewCartServlet to show the updates to the shopping cart.

```xml
<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:output method = "xml" omit-xml-declaration = "no"
doctype-system = "http://www.wapforum.org/DTD/wml_1.1.xml"
doctype-public = "-//WAPFORUM//DTD WML 1.1//EN"/>
<xsl:include href = "/XSLT/WML/error.xsl"/>
<xsl:template match = "cart">
<wml>
  <card title = "Shopping Cart">
    <do type = "prev">
      <prev/>
    </do>
  </card>
</wml>
</xsl:stylesheet>
```

**Fig. 18.10** ViewCartServlet XSL transformation for WML browsers (**WML/viewCart.xsl**) (part 1 of 3). (Image © 2001 Nokia Mobile Phones.)
<do type="accept" label="Check Out">
  <go href="Checkout" method="post"/>
</do>

<p><em>Shopping Cart</em></p>

<p>Your total: $<xsl:value-of select="@total"/></p>

<table columns="2">
  <tr>
    <td>Title</td>
    <td>Price</td>
  </tr>
  <xsl:for-each select="orderProduct">
    <tr>
      <td>
        <a href="#ISBN{product/ISBN}">
          <xsl:value-of select="product/title"/>
        </a>
      </td>
      <td>$<xsl:value-of select="product/price"/></td>
    </tr>
  </xsl:for-each>
</table>

<xsl:for-each select="orderProduct">
  <card id="ISBN{product/ISBN}">
    <do label="OK" type="prev"><prev/></do>
    <do label="Change Quant" type="options">
      <go href="#quant{product/ISBN}"/>
    </do>
    <p><xsl:value-of select="product/title"/></p>
    <p>Quantity: <xsl:value-of select="quantity"/></p>
    <p><xsl:value-of select="product/author"/></p>
  </card>
</xsl:for-each>

<xsl:for-each select="orderProduct">
  <card id="quant{product/ISBN}">
    <p>Enter new quantity</p>
    <input name="quantity" emptyok="false" type="text" format="*n"/>
  </card>
</xsl:for-each>

Fig. 18.10 ViewCartServlet XSL transformation for WML browsers (WML/viewCart.xsl) (part 2 of 3). (Image © 2001 Nokia Mobile Phones.)
1036 Enterprise Java Case Study: Presentation and Controller Logic Chapter 18

// RemoveFromCartServlet.java
// RemoveFromCartServlet removes a Product from the Customer's ShoppingCart.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;

Fig. 18.11 RemoveFromCartServlet for removing products from shopping cart (part 1 of 3).

Fig. 18.10 ViewCartServlet XSL transformation for WML browsers (WML/viewCart.xsl) (part 3 of 3). (Image © 2001 Nokia Mobile Phones.)
// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public class RemoveFromCartServlet extends XMLServlet {

    // respond to HTTP post requests
    public void doPost( HttpServletRequest request,
                        HttpServletResponse response )
                      throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();
        // remove Product from ShoppingCart
        try {
            // get ISBN of Product to be removed
            String isbn = request.getParameter( "ISBN" );
            // get Customer's ShoppingCart from session
            HttpSession session = request.getSession();
            ShoppingCart shoppingCart =
                ( ShoppingCart ) session.getAttribute( "cart" );
            // if ShoppingCart is not null, remove Product
            if ( shoppingCart != null )
                shoppingCart.removeProduct( isbn );
            // redirect Customer to ViewCartServlet to view
            // the contents of ShoppingCart
            response.sendRedirect( "ViewCart" );
        } // end try

        // handle exception if Product not found in ShoppingCart
        catch ( ProductNotFoundException productException ) {
            productException.printStackTrace();
            // append error message to XML document
            document.appendChild( buildErrorMessage( document, productException.getMessage() ) );
        }
    }

Fig. 18.11 RemoveFromCartServlet for removing products from shopping cart (part 2 of 3).
```java
// UpdateCartServlet.java
// UpdateCartServlet updates the quantity of a Product in the
// Customer's ShoppingCart.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public class UpdateCartServlet extends XMLServlet {

    // respond to HTTP post requests
    public void doPost( HttpServletRequest request,
                        HttpServletResponse response )
                        throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();

        // update quantity of given Product in ShoppingCart
        try {
            // get Customer's ShoppingCart from session
            HttpSession session = request.getSession( false );
            ShoppingCart shoppingCart = ( ShoppingCart )
                                         session.getAttribute( "cart" );

            // get Enumeration of parameter names
            Enumeration parameters = request.getParameterNames();
```

Fig. 18.12 *UpdateCartServlet* for updating quantities of products in shopping cart (part 1 of 2).
Method `setProductQuantity` throws a `ProductNotFoundException` if the product with the given ISBN is not found in the shopping cart. Lines 66–73 handle this exception and generate an XML error message using method `buildErrorMessage`. The method `writeXML` presents the error message to the client (line 72).

### 18.3.5 CheckoutServlet

Once the customer has finished browsing through the store and adding products to the shopping cart, `CheckoutServlet` (Fig. 18.13) completes the customer’s Order.

A customer must log into the on-line store before checking out. Lines 32–33 retrieve the customer’s `userID` from the `HttpSession`. Lines 39–40 then obtain a remote reference to the customer’s `ShoppingCart` from the `session` object. After ensuring that neither the `ShoppingCart` nor the `userID` is `null`, line 51 invokes `ShoppingCart`
method `checkout` to place the order. Method `checkout` returns a remote reference to an `Order` EJB instance, which represents the customer’s order. Line 54 gets the `orderID` for the `Order` by invoking method `getPrimaryKey`. Lines 57–58 redirect the client to `ViewOrderServlet`, which displays the details of the order that was placed.

If the `userID` is `null`, lines 60–70 generates an error message indicating that the customer is not logged in. Lines 75–83 catch a `ProductNotFoundException`, which method `checkout` throws if the shopping cart is empty (i.e., contains no products).

18.4 Product Catalog Servlets

Servlets `GetAllProductsServlet` and `ProductSearchServlet` provide an online catalog. These servlets retrieve a list of products and present the list to the customer. `GetProductServlet` shows the details of a given product.

18.4.1 `GetAllProductsServlet`

`GetAllProductsServlet` (Fig. 18.14) provides a list of products available at our online store. Lines 35–44 retrieve a remote reference to the `Product` EJB, which represents a product in the store. Method `findAllProducts` of interface `ProductHome` returns a `Collection` of `Product` EJBs, each of which represents a single product (lines 47–48). Lines 58–70 iterate through the `Collection` of products to build an XML document that contains information about each product. Lines 64–65 obtain a `ProductModel` for each product and lines 68–69 retrieve each product’s XML description.
throws ServletException, IOException
{
    Document document = getDocumentBuilder().newDocument();
    HttpSession session = request.getSession();
    // get Customer's userID from session
    String userID = (String) session.getAttribute("userID");
    // get ShoppingCart and check out
    try {
        ShoppingCart shoppingCart = (ShoppingCart) session.getAttribute("cart");
        // ensure Customer has a ShoppingCart
        if (shoppingCart == null)
            throw new ProductNotFoundException("Your " +
                "ShoppingCart is empty." );
        // ensure userID is neither null nor empty
        if (!(userID == null || userID.equals(""))) {
            // invoke checkout method to place Order
            Order order = shoppingCart.checkout(userID);
            // get orderID for Customer's Order
            Integer orderID = (Integer) order.getPrimaryKey();
            // go to ViewOrder to show completed order
            response.sendRedirect("ViewOrder?orderID=" +
                orderID);
        } else {
            // userID was null, indicating Customer is
            // not logged in
            String error = "You are not logged in."
            // append error message to XML document
            document.appendChild(buildErrorMessage(document, error));
            writeXML(request, response, document);  
        }
    } // end try

    // handle exception if Product not found in ShoppingCart
    catch (ProductNotFoundException productException) {
        productException.printStackTrace();
    }

Fig. 18.13 CheckoutServlet for placing Orders (part 2 of 3). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
1042 Enterprise Java Case Study: Presentation and Controller Logic Chapter 18

```java
// append error message to XML document
document.appendChild(buildErrorMessage(document, productException.getMessage()));

writeXML(request, response, document);

} // end method doPost
```

Fig. 18.13 CheckoutServlet for placing Orders (part 3 of 3). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
// GetAllProductsServlet.java
// GetAllProductsServlet retrieves a list of all Products
// available in the store and presents the list to the client.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.rmi.*;
import javax.naming.*;
import javax.ejb.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;

public class GetAllProductsServlet extends XMLServlet {
    // respond to HTTP get requests
    public void doGet( HttpServletRequest request,
        HttpServletResponse response )
        throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();

        // generate Product catalog
        try {
            InitialContext context = new InitialContext();

            // look up Product EJB
            Object object =
                context.lookup( "java:comp/env/ejb/Product" );

            // get ProductHome interface to find all Products
            ProductHome productHome = ( ProductHome )
                PortableRemoteObject.narrow( object,
                ProductHome.class );

            // get Iterator for Product list
            Iterator products =
                productHome.findAllProducts().iterator();

            // create root of XML document
            Element rootElement =
                document.createElement( "catalog" );
        }
    }
}
// append catalog Element to XML document
document.appendChild( rootElement );

// add each Product to the XML document
while ( products.hasNext() ) {
    Product product = ( Product )
        PortableRemoteObject.narrow( products.next(),
        Product.class );

    // get ProductModel for current Product
    ProductModel productModel =
        product.getProductModel();

    // add an XML element to document for Product
    rootElement.appendChild( productModel.getXML( document ) );
}

} // end try

// handle exception when looking up Product EJB
catch ( NamingException namingException ) {
    namingException.printStackTrace();

    String error = "The Product EJB was not found in " +
        "the JNDI directory."

    // append error message to XML document
    document.appendChild( buildErrorMessage( 
        document, error ) );
}

// handle exception when a Product cannot be found
catch ( FinderException finderException ) {
    finderException.printStackTrace();

    String error = "No Products found in the store."

    // append error message to XML document
    document.appendChild( buildErrorMessage( 
        document, error ) );
}

// ensure content is written to client
finally {
    writeXML( request, response, document );
}
} // end method doGet

Fig. 18.14 GetAllProductsServlet for viewing the product catalog (part 2 of
3). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
Fig. 18.14 *GetAllProductsServlet* for viewing the product catalog (part 3 of 3). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
Method **lookup** (line 39) throws a **NamingException** if the **Product** EJB cannot be found in the JNDI directory. The corresponding **catch** block (lines 75–84) produces an XML error message indicating that the JNDI lookup of the **Product** EJB failed. Method **findAllProducts** (line 48) throws a **FinderException** if no **Product**s are found. Lines 87–95 **catch** this exception and build an XML error message that indicates there were no products found in the database. The **finally** block (lines 98–100) presents the content to the client using method **writeXML**.

### 18.4.2 GetProductServlet

**GetAllProductsServlet** presents a list of all **Product**s in the store. To view the details of a given product, the customer invokes **GetProductServlet** (Fig. 18.15). This servlet uses the application’s business logic to retrieve detailed information about a given product for the customer.

```java
public class GetProductServlet extends XMLServlet {
    public void doGet( HttpServletRequest request,
                       HttpServletResponse response )
                       throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();
        // get ISBN from request object
        String isbn = request.getParameter( "ISBN" );
        // generate XML document with Product details
```

---

Fig. 18.15 **GetProductServlet** for viewing product details (part 1 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
try {
    InitialContext context = new InitialContext();

    // look up Product EJB
    Object object =
        context.lookup( "java:comp/env/ejb/Product" );

    // get ProductHome interface to find Product
    ProductHome productHome = ( ProductHome )
        PortableRemoteObject.narrow(
            object, ProductHome.class );

    // find Product with given ISBN
    Product product =
        productHome.findByPrimaryKey( isbn );

    // create XML document root Element
    Node rootNode =
        document.createElement( "bookstore" );

    // append root Element to XML document
    document.appendChild( rootNode );

    // get Product details as a ProductModel
    ProductModel productModel =
        product.getProductModel();

    // build an XML document with Product details
    rootNode.appendChild( productModel.getXML( document ) );
}

// handle exception when looking up Product EJB
catch ( NamingException namingException ) {
    namingException.printStackTrace();

    String error = "The Product EJB was not found in " +
        "the JNDI directory."

    document.appendChild( buildErrorMessage( document, error ) );
}

// handle exception when Product is not found
catch ( FinderException finderException ) {
    finderException.printStackTrace();

    String error = "The Product with ISBN " + isbn +
        " was not found in our store.";

Fig. 18.15 GetProductServlet for viewing product details (part 2 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
```java
    document.appendChild( buildErrorMessage(
        document, error ) );

    // ensure content is written to client
    finally {
        writeXML( request, response, document );
    }

    } // end method doGet

Fig. 18.15 GetProductServlet for viewing product details (part 3 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
The **Product** EJB provides the details of a given product as a **ProductModel**. Lines 39–45 retrieve a reference to the **ProductHome** interface. Lines 48–49 invoke **ProductHome** interface method **findByPrimaryKey** to obtain the **Product** EJB for the given **ISBN**. Lines 59–64 obtain the details of the product as a **ProductModel** and use method **getXML** to generate the XML content for the client.

Lines 69–77 catch a **NamingException** if the **lookup** of the **ProductHome** interface fails. Lines 80–88 catch a **FinderException**, which is generated by the call to **findByPrimaryKey** on line 49 if a product with the given **ISBN** is not found in the database. In each catch block, we create an XML error message to inform the user of the error. The finally block on lines 91–93 ensures that the content is presented to the client, regardless of any exceptions that may be thrown.

### 18.4.3 ProductSearchServlet

**ProductSearchServlet** (Fig. 18.16) searches the database for products whose titles contain a particular keyword. Method **findByTitle** of the **Product** EJB’s home interface takes a **String** as an argument and finds a product whose title contains the provided keyword. Because a search of this type may match many different products, method **findByTitle** returns an **Collection** of **Product** EJB remote references.
Lines 34–35 get the `searchString` parameter from the request object and place a wildcard character (%) before and after the `searchString`. The database uses this wildcard character to match titles that contain the given `searchString`. Lines 50–51 create an `Iterator` to iterate through the list of products returned by the search. Lines 58–69 add an XML representation to the XML document for each product found.

```java
// ProductSearchServlet.java
// ProductSearchServlet allows a Customer to search through
// the store for a particular Product.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;

public class ProductSearchServlet extends XMLServlet {

    // respond to HTTP get requests
    public void doGet( HttpServletRequest request,
    HttpServletResponse response )
    throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();

        // get the searchString from the request object
        String searchString = "%" +
            request.getParameter( "searchString" ) + "%";

        // find Product using Product EJB
        try {
            InitialContext context = new InitialContext();

            // look up Product EJB
            Object object =
                context.lookup( "java:comp/env/ejb/Product" );

            ProductHome productHome = ( ProductHome )
```
PortableRemoteObject.narrow(
  object, ProductHome.class );

// find Products that match searchString
Iterator products = productHome.findByTitle( searchString ).iterator();

// create catalog document element
Node rootNode = document.appendChild(
  document.createElement( "catalog" ) );

// generate list of matching products
while ( products.hasNext() ) {
  Product product = ( Product )
    PortableRemoteObject.narrow( products.next(),
    Product.class );

  ProductModel productModel = product.getProductModel();

  // append XML element to the document for the
  // current Product
  rootNode.appendChild( productModel.getXML( document ) );
}

} // end try

// handle exception when looking up Product EJB
try ( NamingException namingException ) {
  namingException.printStackTrace();

  String error = "The Product EJB was not found in " +
  "the JNDI directory."

  document.appendChild( buildErrorMessage( document, error ) );
}

// handle exception when Product is not found
try ( FinderException finderException ) {
  finderException.printStackTrace();

  String error = "No Products match your search."

  document.appendChild( buildErrorMessage( document, error ) );
}

// ensure content is written to client
try {
  writeXML( request, response, document );
}

Fig. 18.16 ProductSearchServlet for searching product catalog (part 2 of 3).
(Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
```java
98     } // end method doGet
99 }
100 }
```

Fig. 18.16 **ProductSearchServlet** for searching product catalog (part 3 of 3).
(Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
18.5 Customer Management Servlets

Online stores typically allow customers to register personal information (e.g., name, e-mail address and shipping address) with the store, for a number of purposes. Primarily, the store needs the information provided in a customer registration to bill the customer and ship products from the store. By registering, a customer will need to enter this information only once. Later, a customer can return to the store and log-in with a user name and password to retrieve previously entered billing and shipping information. Customer registrations also allow on-line stores to provide customers with personalized content and order-tracking information. Some on-line stores also can use customer registration information to help target advertisements to customers with certain preferences or within particular demographics.

RegisterServlet (Section 18.5.1) handles customer registrations for the Deitel Bookstore. Once a customer registers, LoginServlet allows the customer to log into the site with a user name and password. ViewOrderHistoryServlet allows customers to see information about orders they already have placed. GetLostPasswordServlet provides Customers with a hint to remind them of forgotten passwords.

18.5.1 RegisterServlet

RegisterServlet (Fig. 18.17) processes the registration forms submitted by new customers. The servlet creates a CustomerModel instance (line 34) and uses the parameter values received from the client to populate the model with details about the customer (lines 38–125). Once the CustomerModel has been populated with data, line 133 looks up the Customer EJB, which represents a customer in the database. Lines 141–142 create a new customer registration in the database by invoking Customer EJB method create with the newly created CustomerModel object as an argument. After registering the customer, lines 147–157 forward the customer to LoginServlet to log into the store.

```java
// RegisterServlet.java
// RegisterServlet processes the Customer registration form
// to register a new Customer.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.*;

// third-party packages
import org.w3c.dom.*;

Fig. 18.17 RegisterServlet for registering new Customers (part 1 of 5).
```
// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;

public class RegisterServlet extends XMLServlet {

    // respond to HTTP post requests
    public void doPost( HttpServletRequest request,
            HttpServletResponse response )
            throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();

        // create CustomerModel to store registration data
        CustomerModel customerModel = new CustomerModel();

        // set properties of CustomerModel using values passed through request object
        customerModel.setUserID( request.getParameter( "userID" ) );

        customerModel.setPassword( request.getParameter( "password" ) );

        customerModel.setPasswordHint( request.getParameter( "passwordHint" ) );

        customerModel.setFirstName( request.getParameter( "firstName" ) );

        customerModel.setLastName( request.getParameter( "lastName" ) );

        // set credit card information
        customerModel.setCreditCardName( request.getParameter( "creditCardName" ) );

        customerModel.setCreditCardNumber( request.getParameter( "creditCardNumber" ) );

        customerModel.setCreditCardExpirationDate( request.getParameter( "creditCardExpirationDate" ) );

        // create AddressModel for billing address
        AddressModel billingAddress = new AddressModel();

        billingAddress.setFirstName( request.getParameter( "billingAddressFirstName" ) );

        billingAddress.setLastName( request.getParameter( "billingAddressLastName" ) );

        billingAddress.setStreetAddressLine1(  

Fig. 18.17 RegisterServlet for registering new Customers (part 2 of 5).
request.getParameter("billingAddressStreet1");

billingAddress.setStreetAddressLine2(request.getParameter("billingAddressStreet2");

billingAddress.setCity(request.getParameter("billingAddressCity");

billingAddress.setState(request.getParameter("billingAddressState");

billingAddress.setZipCode(request.getParameter("billingAddressZipCode");

billingAddress.setCountry(request.getParameter("billingAddressCountry");

billingAddress.setPhoneNumber(request.getParameter("billingAddressPhoneNumber");

customerModel.setBillingAddress(billingAddress);

// create AddressModel for shipping address
AddressModel shippingAddress = new AddressModel();

shippingAddress.setFirstName(request.getParameter("shippingAddressFirstName");

shippingAddress.setLastName(request.getParameter("shippingAddressLastName");

shippingAddress.setStreetAddressLine1(request.getParameter("shippingAddressStreet1");

shippingAddress.setStreetAddressLine2(request.getParameter("shippingAddressStreet2");

shippingAddress.setCity(request.getParameter("shippingAddressCity");

shippingAddress.setState(request.getParameter("shippingAddressState");

shippingAddress.setZipCode(request.getParameter("shippingAddressZipCode");

shippingAddress.setCountry(request.getParameter("shippingAddressCountry");

shippingAddress.setPhoneNumber(request.getParameter("shippingAddressPhoneNumber");

customerModel.setShippingAddress(shippingAddress);

Fig. 18.17 RegisterServlet for registering new Customers (part 3 of 5).
try {
    InitialContext context = new InitialContext();

    Object object = context.lookup( "java:comp/env/ejb/Customer" );

    CustomerHome customerHome = ( CustomerHome ) PortableRemoteObject.narrow( object, CustomerHome.class );

    Customer customer = customerHome.create( customerModel );

    customerModel = customer.getCustomerModel();

    RequestDispatcher dispatcher = getServletContext().getRequestDispatcher( "/Login" );

    request.setAttribute( "userID", customerModel.getUserID() );
    request.setAttribute( "password", customerModel.getPassword() );

    dispatcher.forward( request, response );
} // end try

// handle exception when looking up Customer EJB
catch ( NamingException namingException ) {
    namingException.printStackTrace();

    String error = "The Customer EJB was not " + "found in the JNDI directory.";

    document.appendChild( buildErrorMessage( document, error ) );
    writeXML( request, response, document );
}

// handle exception when creating Customer
catch ( CreateException createException ) {
    createException.printStackTrace();

    String error = "The Customer could not be created";

Fig. 18.17 RegisterServlet for registering new Customers (part 4 of 5).
Lines 162–172 catch a NamingException, which indicates that the CustomerHome interface could not be found in the JNDI directory. Because RegisterServlet creates a new Customer, lines 175–184 catch a CreateException in case the Customer EJB could not be created. Each catch block builds an XML error message using method buildErrorMessage and invokes method writeXML to display the error message to the customer.

### 18.5.2 LoginServlet

To log into the store, a registered customer must provide a valid **userID** and **password**, which the client submits to LoginServlet. LoginServlet (Fig. 18.18) checks the **userID** and **password** against the **userIDs** and **passwords** stored in the database.
```java
// respond to HTTP post requests
public void doPost( HttpServletRequest request,
    HttpServletResponse response )
    throws ServletException, IOException
{
    Document document = getDocumentBuilder().newDocument();

    String userID = request.getParameter( "userID" );
    String password = request.getParameter( "password" );

    // use Customer EJB to authenticate user
    try {
        InitialContext context = new InitialContext();

        // look up Customer EJB
        Object object =
            context.lookup( "java:comp/env/ejb/Customer" );

        CustomerHome customerHome = ( CustomerHome )
            PortableRemoteObject.narrow( object,
            CustomerHome.class );

        // find Customer with given userID and password
        Customer customer =
            customerHome.findByLogin( userID, password );

        // get CustomerModel for Customer
        CustomerModel customerModel =
            customer.getCustomerModel();

        // set userID in Customer's session
        request.getSession().setAttribute( "userID",
            customerModel.getUserID() );

        // create login XML element
        Element login = document.createElement( "login" );
        document.appendChild( login );

        // add Customer's first name to XML document
        Element firstName =
            document.createElement( "firstName" );
        firstName.appendChild( document.createTextNode(
            customerModel.getFirstName() ) );
        login.appendChild( firstName );
    } // end try

    // handle exception when looking up Customer EJB
    catch ( NamingException namingException ) {
        namingException.printStackTrace();
    }
}
```
String error = "The Customer EJB was not found in " +
"the JNDI directory."

document.appendChild( buildErrorMessage(
    document, error ) );

// handle exception when Customer is not found
catch ( FinderException finderException ) {
    finderException.printStackTrace();
    String error = "The userID and password entered " +
    "were not found."
    document.appendChild( buildErrorMessage(
        document, error ) );
}

// ensure content is written to client
finally {
    writeXML( request, response, document );
}

} // end method doPost

Fig. 18.18 LoginServlet for authenticating registered Customers (part 3 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
LoginServlet uses the Customer EJB to validate the userID and password that the customer entered. Lines 39–44 obtain a reference to the CustomerHome interface. Lines 47–48 invoke CustomerHome method findByLogin, which returns a remote reference to the Customer with the userID and password that the user provided. Once the Customer is found, lines 59–69 build a simple XML document that indicates the customer has successfully logged into the store.

Lines 74–82 catch a NamingException if the Customer EJB cannot be found in the JNDI directory. If no Customer is found that matches the userID and password the user entered, lines 85–93 catch a FinderException. Each catch block builds an XML error message for the client to display. Lines 96–98 present the content to the client.

18.5.3 ViewOrderHistoryServlet

Registered customers may want to see information about orders they have placed in the past. ViewOrderHistoryServlet (Fig. 18.19) allows customers to see orders they have placed, along with the dates the orders were taken, the total costs of the orders and whether or not the orders were shipped from the warehouse.
// ViewOrderHistoryServlet.java
// ViewOrderHistoryServlet presents a list of previous Orders
// to the Customer.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;
import java.util.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public class ViewOrderHistoryServlet extends XMLServlet {

   // respond to HTTP get requests
   public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
       throws ServletException, IOException
   {
      Document document = getDocumentBuilder().newDocument();

      HttpSession session = request.getSession();
      String userID = ( String )
         session.getAttribute( "userID" );

      // build order history using Customer EJB
      try {
         InitialContext context = new InitialContext();

         // look up Customer EJB
         Object object =
            context.lookup( "java:comp/env/ejb/Customer" );

         CustomerHome customerHome = ( CustomerHome )
            PortableRemoteObject.narrow(
               object, CustomerHome.class );

         // find Customer with given userID
         Customer customer =

Fig. 18.19 ViewOrderHistoryServlet for viewing customer’s previously placed Orders (part 1 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
customerHome.findByUserID( userID );

// create orderHistory element
Element rootNode = ( Element ) document.appendChild(
    document.createElement( "orderHistory" ) );

// get Customer's Order history
Iterator orderHistory =
    customer.getOrderHistory().iterator();

// loop through Order history and add XML elements
// to XML document for each Order
while ( orderHistory.hasNext() ) {
    OrderModel orderModel =
        ( OrderModel ) orderHistory.next();

    rootNode.appendChild(
        orderModel.getXML( document ) );
}

// handle exception when Customer has no Order history
catch ( NoOrderHistoryException historyException ) {
    historyException.printStackTrace();
    document.appendChild( buildErrorMessage( document,
        historyException.getMessage() ) );
}

// handle exception when looking up Customer EJB
catch ( NamingException namingException ) {
    namingException.printStackTrace();
    String error = "The Customer EJB was not found in " +
        "the JNDI directory."
    document.appendChild( buildErrorMessage( document,
        error ) );
}

// handle exception when Customer is not found
catch ( FinderException finderException ) {
    finderException.printStackTrace();
    String error = "The Customer with userID " + userID +
        " was not found."
    document.appendChild( buildErrorMessage( document,
        error ) );
}

Fig. 18.19 ViewOrderHistoryServlet for viewing customer’s previously placed Orders (part 2 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
// ensure content is written to client
finally {
    writeXML(request, response, document);
}
} // end method doGet

Fig. 18.19 ViewOrderHistoryServlet for viewing customer’s previously placed Orders (part 3 of 4). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
Customer EJB method `getOrderHistory` returns a `Collection` of the customer’s previous orders. Lines 51–52 obtain the Customer EJB for the Customer, who must be logged into the bookstore. Lines 59–60 retrieve an `Iterator` for the customer’s order history. Lines 64–70 loop through the order history and build the XML document to present to the client.

If the customer has not placed any orders in our on-line store, method `getOrderHistory` throws a `NoOrderHistoryException`. Lines 74–79 catch this exception and build an error message to display to the customer. If the `CustomerHome` interface could not be found or the `Customer` could not be found in the database, a `NamingException` or `FinderException` is thrown, respectively. Lines 82–101 catch each of these exceptions and construct an error message, using method `buildErrorMessage`. Line 105 presents the content to the client, using method `writeXML`.

### 18.5.4 `ViewOrderServlet`

`ViewOrderServlet` (Fig. 18.20) displays the details of an order. `CheckoutServlet` forwards clients to `ViewOrderServlet` when a customer places an order. `ViewOrderHistoryServlet` forwards clients to `ViewOrderServlet` to present the details of an order that has already been placed.
Lines 39–44 obtain a reference to interface `OrderHome`. Lines 47–48 retrieve the `orderID` parameter from the `request` object. Line 51 invokes `OrderHome` method `findByPrimaryKey` to obtain a remote reference to the `Order` with the given `orderID`. Lines 54–58 get the `OrderModel` for the `Order` and append its XML representation to the XML document.

Lines 63–71 catch a `NamingException`, which is thrown from method `lookup` if the `Order` EJB cannot be found in the JNDI directory. Lines 74–82 catch a `FinderException`, which is thrown by method `findByPrimaryKey` if an `Order` with the given `orderID` is not found. Line 86 presents the XML document to the client, using method `writeXML`.

```java
// ViewOrderServlet.java
// ViewOrderServlet presents the contents of a Customer's Order.
package com.deitel.advjhtp1.bookstore.servlets;

// Java core packages
import java.io.*;

// Java extension packages
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.*;

// third-party packages
import org.w3c.dom.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.ejb.*;

public class ViewOrderServlet extends XMLServlet {
    // respond to HTTP get requests
    public void doGet( HttpServletRequest request,
                        HttpServletResponse response )
        throws ServletException, IOException
    {
        Document document = getDocumentBuilder().newDocument();
        Integer orderID = null;

        // look up Order EJB and get details of Order with
        // given orderID
        try {
            InitialContext context = new InitialContext();
            // look up Order EJB
```
Object object =
    context.lookup( "java:comp/env/ejb/Order" );

OrderHome orderHome = ( OrderHome )
    PortableRemoteObject.narrow( object, OrderHome.class );

// get orderID from request object
orderID = new Integer(
    request.getParameter( "orderID" ) );

// find Order with given orderID
Order order = orderHome.findByPrimaryKey( orderID );

// get Order details as an OrderModel
OrderModel orderModel = order.getOrderModel();

// add Order details to XML document
document.appendChild( orderModel.getXML( document ) );

// handle exception when looking up Order EJB
catch ( NamingException namingException ) {
    namingException.printStackTrace();

    String error = "The Order EJB was not found in " +
                   "the JNDI directory.";

    document.appendChild( buildErrorMessage( document, error ) );
}

// handle exception when Order is not found
catch ( FinderException finderException ) {
    finderException.printStackTrace();

    String error = "An Order with orderID " + orderID +
                   " was not found.";

    document.appendChild( buildErrorMessage( document, error ) );
}

// ensure content is written to client
finally {
    writeXML( request, response, document );
}

} // end method doGet

Fig. 18.20 ViewOrderServlet for viewing details of an order (part 2 of 3). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
Registered Customers occasionally forget their passwords. **GetPasswordHintServlet** (Fig. 18.21) provides hints to help Customers remember their passwords. The customer supplies the password hint as part of the registration process.
The hint is stored with other customer registration information in the Customer EJB. Lines 38–47 look up the CustomerHome interface and retrieve the Customer EJB remote reference. Customer EJB method getPasswordHint (line 55) returns the hint the user entered when registering on the site. Line 58 adds the hint to the XML document.

In this chapter, we presented the controller and presentation logic for the Deitel Bookstore. This controller logic provides an HTTP interface to the business logic objects we present in Chapters 18 and 19. Java servlets provide a robust and flexible controller logic implementation. XSLT presentation logic allows the Deitel Bookstore application to support many different client types without a need for changes in controller logic implementations. In Chapters 19 and 20, we present the business logic for the Deitel Bookstore, using Enterprise JavaBeans.
context.lookup( "java:comp/env/ejb/Customer" );

CustomerHome customerHome = ( CustomerHome )
PortableRemoteObject.narrow( object,
    CustomerHome.class );

// find Customer with given userID
Customer customer =
customerHome.findByUserID( userID );

// create passwordHint element in XML document
Element hintElement =
document.createElement( "passwordHint" );

// add text of passwordHint to XML element
hintElement.appendChild( document.createTextNode( customer.getPasswordHint() ) );

// append passwordHint element to XML document
document.appendChild( hintElement );

} // end try

// handle exception when looking up Customer EJB
catch ( NamingException namingException ) {
namingException.printStackTrace();

String error = "The Customer EJB was not found in " +
    "the JNDI directory."

document.appendChild( buildErrorMessage( document, error ) );
}

// handle exception when Customer is not found
catch ( FinderException finderException ) {
finderException.printStackTrace();

String error = "No customer was found with userID " +
    userID + ".";

document.appendChild( buildErrorMessage( document, error ) );
}

// ensure content is written to client
finally {
    writeXML( request, response, document );
}
} // end method doGet

Fig. 18.21 GetPasswordHintServlet for viewing a Customer’s password hint
(part 2 of 3). (Images courtesy Pixo, Inc. or © 2001 Nokia Mobile Phones.)
SELF-REVIEW EXERCISES

18.1 Which part of the MVC architecture do the servlets in the Deitel Bookstore implement? Which part of the MVC architecture do the XSL transformations implement?
18.2 Write a code snippet for looking up the ShoppingCart EJB in the JNDI directory and creating a new instance using interface ShoppingCartHome. Be sure to catch any exceptions thrown when looking up the EJB or creating a new instance.

18.3 How does ViewOrderServlet (Fig. 18.20) locate the Order that the user requested to view?

18.4 What common functionality does class XMLServlet (Fig. 18.1) provide for the servlets in the Deitel Bookstore? Describe the purposes of the main methods of class XMLServlet.

18.5 How does class XMLServlet determine the name of the XSLT stylesheet to use when transforming content generated by the servlet? What benefit does this strategy provide?

18.6 How does class XMLServlet determine the particular XSLT stylesheet to use when transforming content generated by the servlet for a particular type of client? What benefit does this strategy provide?

ANSWERS TO SELF-REVIEW EXERCISES

18.1 The servlets implement the controller in the MVC architecture, because they handle all user requests and process user input. The XSL transformations implement the view in the MVC design pattern because they produce presentations of application data.

18.2 The following code snippet looks up the ShoppingCart EJB in the JNDI directory and creates a new instance using interface ShoppingCartHome:

```java
try {
    InitialContext context = new InitialContext;

    Object object = context.lookup("java:comp/env/ejb/ShoppingCart");

    ShoppingCartHome shoppingCartHome =
        (ShoppingCartHome) PortableRemoteObject.narrow(
            object, ShoppingCartHome.class);

    ShoppingCart shoppingCart = shoppingCartHome.create();
}
catch (NamingException namingException) {
    namingException.printStackTrace();
}
catch (CreateException createException) {
    createException.printStackTrace();
}
```

18.3 ViewOrderServlet uses OrderHome method findByPrimaryKey to locate the Order, with the orderID passed as a parameter to the HttpServletRequest object.

18.4 Class XMLServlet provides a common init method for initializing the DocumentBuilderFactory, TransformerFactory and properties each servlet uses. Class XMLServlet provides method buildErrorMessage, which creates an XML element to describe an error message. Class XMLServlet also provides method writeXML, which uses method transform to transform the XML content generated by each servlet using client-specific XSLT transformations.
18.5 Class `XMLServlet` has property `XSLFileName` that specifies the name of the XSL file to use when transforming the servlet’s content. Class `XMLServlet`’s `init` method sets the `XSL-FileName` property to the value specified in the `XSL_FILE` servlet initialization parameter. Determining the file name from an initialization parameter enables the deployer to specify the file name when deploying the application. The file name can be changed later without the need to recompile the servlet.

18.6 Class `XMLServlet` uses a `ClientModel` to determine which directory contains the XSL transformation for XML content generated by the servlet. Class `XMLServlet` creates a list of `ClientModels` from an XML configuration file when the servlet is first initialized. Each `ClientModel` specifies a `User-Agent` header that uniquely identifies the client, the `Content-Type` for sending data to the client and the directory in which the XSL transformations can be found for generating content specific to the client. This enables the developer to add support for new client types without modifying any servlet code. The developer simply provides a set of XSL transformations for the new client type and includes information about the new client type in `clients.xml`. 
Objectives

• To understand the EJB data model for the Deitel Bookstore case study.
• To understand the business logic used in the Deitel Bookstore case study.
• To understand performance issues involved in transmitting objects over RMI-IIOP.
• To understand the benefits of EJBs that use container-managed persistence for database storage.
• To understand the usage of primary-key classes that represent complex primary keys.

Drive thy business, or it will drive thee.
Benjamin Franklin

Everybody’s business is nobody’s business, and nobody’s business is my business.
Clara Barton
19.1 Introduction

In this chapter, we present the EJB business logic for the shopping-cart e-commerce model, and entity EJBs that provide an object-based interface to the store’s product catalog. After reading this chapter, you will understand the use of EJBs in an e-commerce application context, as well as more advanced EJB topics, such as custom primary-key classes and many-to-many relationships.

19.2 EJB Architecture

EJBs implement the business logic of the Deitel Bookstore case study. Servlet controller logic communicates with EJB business logic to process user requests and retrieve data from the database. For example, GetProductServlet handles Customer requests to view Product details. GetProductServlet uses the JNDI directory to locate the Product EJB’s home interface. GetProductServlet invokes ProductHome method findByPrimaryKey to retrieve a remote reference to the Product with the requested
ISBN. GetProductServlet must then use methods in the Product remote interface to retrieve information about the Product. Each method call on the Product remote interface incurs network traffic, because communication with the EJB is performed over RMI-IIOP (discussed in Chapter 27). If separate method calls were required to retrieve the Product’s title, author, price and other properties, the network overhead would severely limit the performance and scalability of the application.

The entity EJBs in the Deitel Bookstore case study alleviate this network congestion by using models to transmit EJB data. A model is a Serializable class that contains all the data for a given EJB. These classes are called models because each model class implements a piece of the model in the MVC architecture. Each entity EJB provides a get method that returns its model representation. For example, the Product EJB has method getProductModel, which returns a ProductModel containing the ISBN, author, price and other properties of a Product. Many of the entity EJBs also provide create methods that accept models as arguments. These create methods create new EJB instances and set the data in the EJB, using property values provided in the model.

Performance Tip 19.1

Aggregating entity EJB data into a model class and returning instances of this model class from EJB business methods can improve EJB performance by reducing the network traffic associated with multiple method calls over RMI-IIOP.

Figure 19.1 shows a sample communication between servlet GetProductServlet and the Product EJB. To get the details of a given Product, GetProductServlet invokes Product method getProductModel. Method getProductModel returns a ProductModel object containing data for a given Product and serializes the ProductModel over RMI-IIOP. GetProductServlet retrieves the ProductModel’s property values to build the output for the user.

19.3 ShoppingCart Implementation

Stateful session EJB ShoppingCart implements business logic for managing each Customer’s shopping cart. The ShoppingCart EJB consists of a remote interface, an EJB implementation and a home interface. We implement ShoppingCart as a stateful session EJB so each ShoppingCart instance will persist throughout a customer’s shopping session. Just as customers of brick-and-mortar stores use shopping carts to gather products, customers of our on-line store use ShoppingCart EJBs to gather products while they browse through our store.

19.3.1 ShoppingCart Remote Interface

Remote interface ShoppingCart (Fig. 19.2) defines business logic methods available in the ShoppingCart EJB. Each remote interface method must declare that it throws RemoteException. Each method also must declare any application-specific exceptions that may be thrown from the implementation. Method getContents (line 20) returns a Collection of Products in the ShoppingCart. Method addProduct (lines 23–24) takes as a String argument the ISBN of a Product to add to the ShoppingCart. Method addProduct throws ProductNotFoundException, which is an application-specific exception that indicates the Product with the given ISBN is not in the database and therefore could not be added to the ShoppingCart.
Method **removeProduct** (lines 27–28) removes the **Product** with the given **ISBN** from the **ShoppingCart**. If the **Product** with the given **ISBN** is not found in the **ShoppingCart**, method **removeProduct** throws a **ProductNotFoundException**.

Method **setProductQuantity** (lines 32–34) updates the quantity of the **Product** with the given **ISBN** in the **ShoppingCart**. For example, if there were one copy of *Advanced Java 2 Platform How to Program* in the **Customer’s Shopping Cart**, **setProductQuantity** could be called with the **ISBN** of *Advanced Java How to Program* and the integer 5 to purchase five copies of the book. If the **Product** with the given **ISBN** is not in the **ShoppingCart**, method **setProductQuantity** throws the application-specific **ProductNotFoundException**.

Method **checkout** (lines 37–38) places an **Order** for the **Products** in the **Customer’s Shopping Cart**. Method **checkout** takes as a **String** argument the **userID** of the customer placing the **Order**. Only registered customers may place **Orders**. Method **getTotal** (line 41) returns the total cost of the **Products** in the **Customer’s Shopping Cart**.

```java
// ShoppingCart.java
// ShoppingCart is the remote interface for stateful session
// EJB ShoppingCart.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.rmi.RemoteException;
import java.util.ArrayList;

// Java extension packages
import javax.ejb.EJBObject;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;
```

**Fig. 19.1** Communication between **GetProductServlet** and **Product EJB**.

**Fig. 19.2** **ShoppingCart** remote interface for adding, removing and updating **Products**, checking out and calculating the **Order**’s total cost (part 1 of 2).
19.3.2 ShoppingCartEJB Implementation

ShoppingCart remote interface implementation ShoppingCartEJB (Fig. 19.3) contains an Collection of OrderProductModels (line 24). An OrderProductModel (Fig. 19.25) represents an item in the ShoppingCart. Each OrderProductModel contains a Product and that Product’s quantity in the ShoppingCart. Method ejbCreate (lines 27–30) initializes the Collection (line 29). Method getContents (line 33–36) returns the contents of the ShoppingCart as a Collection of OrderProductModels.
```java
import java.text.SimpleDateFormat;

// Java extension packages
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public class ShoppingCartEJB implements SessionBean {
    private SessionContext sessionContext;

    // OrderProductModels (Products & quantities) in ShoppingCart
    private Collection orderProductModels;

    // create new ShoppingCart
    public void ejbCreate()
    {
        orderProductModels = new ArrayList();
    }

    // get contents of ShoppingCart
    public Collection getContents()
    {
        return orderProductModels;
    }

    // add Product with given ISBN to ShoppingCart
    public void addProduct(String isbn)
    throws ProductNotFoundException, EJBException
    {
        // check if Product with given ISBN is already
        // in ShoppingCart
        Iterator iterator = orderProductModels.iterator();

        while (iterator.hasNext()) {
            OrderProductModel orderProductModel =
            (OrderProductModel)iterator.next();

            ProductModel productModel =
            orderProductModel.getProductModel();

            // if Product is in ShoppingCart, increment quantity
            if (productModel.getISBN().equals(isbn)) {
                orderProductModel.setQuantity(
                orderProductModel.getQuantity() + 1);
            }
        }
    }
```

**Fig. 19.3** *ShoppingCartEJB* implementation of *ShoppingCart* remote interface (part 2 of 7).
Fig. 19.3  ShoppingCartEJB implementation of ShoppingCart remote interface (part 3 of 7).
{ 
    Iterator iterator = orderProductModels.iterator();

    while ( iterator.hasNext() ) {

        // get next OrderProduct in ShoppingCart
        OrderProductModel orderProductModel =
            ( OrderProductModel ) iterator.next();

        ProductModel productModel =
            orderProductModel.getProductModel();

        // remove Product with given ISBN from ShoppingCart
        if ( productModel.getISBN().equals( isbn ) ) {
            orderProductModels.remove( orderProductModel );
            return;
        }
    } // end while

    // throw exception if Product not found in ShoppingCart
    throw new ProductNotFoundException( "The Product " +
        "with ISBN " + isbn + " was not found in your " +
        "ShoppingCart." );
} // end method removeProduct

// set quantity of Product in ShoppingCart
public void setProductQuantity( String isbn,
int productQuantity ) throws ProductNotFoundException {

    // throw IllegalArgumentException if quantity not valid
    if ( productQuantity < 0 )
        throw new IllegalArgumentException( "Quantity cannot be less than zero." );

    // remove Product if productQuantity less than 1
    if ( productQuantity == 0 ) {
        removeProduct( isbn );
        return;
    }

    Iterator iterator = orderProductModels.iterator();

    while ( iterator.hasNext() ) {

        // get next OrderProduct in ShoppingCart
        OrderProductModel orderProductModel =
            ( OrderProductModel ) iterator.next();

        ProductModel productModel =
            orderProductModel.getProductModel();

        // set quantity to given Product
        orderProductModel.setProductQuantity( productQuantity );

        // throw exception if quantity not valid
        if ( productQuantity < 0 )
            throw new IllegalArgumentException( "Quantity cannot be less than zero." );

        // remove Product if productQuantity less than 1
        if ( productQuantity == 0 ) {
            removeProduct( isbn );
            return;
        }

    } // end while

    // throw exception if Product not found in ShoppingCart
    throw new ProductNotFoundException( "The Product " +
        "with ISBN " + isbn + " was not found in your " +
        "ShoppingCart." );
} // end method setProductQuantity

Fig. 19.3 ShoppingCartEJB implementation of ShoppingCart remote interface (part 4 of 7).
orderProductModel.getProductModel();

    // set quantity for Product with given ISBN
    if ( productModel.getISBN().equals( isbn ) ) {
        orderProductModel.setQuantity( productQuantity );
        return;
    }

} // end while

    // throw exception if Product not found in ShoppingCart
    throw new ProductNotFoundException( "The Product " +
        "with ISBN " + isbn + " was not found in your " +
        "ShoppingCart." );

} // end method setProductQuantity

    // checkout of store (i.e., create new Order)
    public Order checkout( String userID )
        throws ProductNotFoundException, EJBException
    {
        // throw exception if ShoppingCart is empty
        if ( orderProductModels.isEmpty() )
            throw new ProductNotFoundException( "There were " +
                "no Products found in your ShoppingCart." );

        // create OrderModel for Order details
        OrderModel orderModel = new OrderModel();

        // set OrderModel's date to today's Date
        orderModel.setOrderDate( new Date() );

        // set list of OrderProduct in OrderModel
        orderModel.setOrderProductModels( orderProductModels );

        // set OrderModel's shipped flag to false
        orderModel.setShipped( false );

        // use OrderHome interface to create new Order
        try {
            InitialContext context = new InitialContext();

            // look up Order EJB
            Object object = context.lookup(
                "java:comp/env/ejb/Order" );

            OrderHome orderHome = ( OrderHome )
                PortableRemoteObject.narrow( object, 
                    OrderHome.class );

            // create new Order using OrderModel and
            // Customer's userID

Fig. 19.3 ShoppingCartEJB implementation of ShoppingCart remote interface (part 5 of 7).
```java
Order order = orderHome.create( orderModel, userID );

// empty ShoppingCart for further shopping
orderProductModels = new ArrayList();

// return Order EJB that was created
return order;
}

// handle exception when looking up Order EJB
} catch ( Exception exception ) {
    throw new EJBException( exception );
}
}

// get total cost for Products in ShoppingCart
public double getTotal()
{
    double total = 0.0;
    Iterator iterator = orderProductModels.iterator();

    // calculate Order's total cost
    while ( iterator.hasNext() ) {

        // get next OrderProduct in ShoppingCart
        OrderProductModel orderProductModel =
            ( OrderProductModel ) iterator.next();

        ProductModel productModel =
            orderProductModel.getProductModel();

        // add OrderProduct extended price to total
        total += ( productModel.getPrice() *
                  orderProductModel.getQuantity() );
    }

    return total;
}

// set SessionContext
public void setSessionContext( SessionContext context )
{
    sessionContext = context;
}

// activate ShoppingCart EJB instance
public void ejbActivate() {}
```

Fig. 19.3  **ShoppingCartEJB** implementation of **ShoppingCart** remote interface (part 6 of 7).
Method `addProduct` (lines 39–108) adds a `Product` to the `ShoppingCart`. Lines 46–62 determine if the `ShoppingCart` already contains the given `Product`. If the `Product` is found in the `ShoppingCart`, lines 56–57 increment the associated `OrderProductModel`'s `quantity`. Otherwise, method `findByPrimaryKey` of interface `ProductHome` locates the `Product` with the given `ISBN` (line 77). Lines 84–85 create an `OrderProductModel` to store the `Product` in the `ShoppingCart`. Line 87 adds the `ProductModel` to the `OrderProductModel`, and line 88 sets the `OrderProductModel`'s `quantity` to 1. Line 91 adds the `OrderProductModel` to the `Collection`, which completes the addition of the `Product` to the `ShoppingCart`.

If method `findByPrimaryKey` of interface `ProductHome` does not find the `Product` with the given primary key, lines 96–101 catch a `FinderException`. Lines 99–100 throw a `ProductNotFoundException` to indicate that a `Product` with the given `ISBN` could not be found.

Method `removeProduct` (lines 111–139) compares the `ISBN` of each `Product` in the `ShoppingCart`’s `OrderProductModel Collection` with the `ISBN` of the `Product` to be removed. If the `Product` with the given `ISBN` is found, line 127 removes the associated `OrderProductModel` from the `Collection`. If the `Product` is not found in the `ShoppingCart`, lines 135–137 throw a `ProductNotFoundException`.

Method `setProductQuantity` (lines 142–180) sets the `quantity` of an `OrderProductModel` in the `ShoppingCart`. If argument `productQuantity` is less than 0, lines 147–148 throw an `IllegalArgumentException`. If the `productQuantity` equals 0, line 152 removes the `Product` from the `ShoppingCart`. Lines 158–173 compare the `ISBN` of each `Product` in the `OrderProductModel Collection` with the given `ISBN`. Line 169 updates the matching `OrderProductModel`'s `quantity` by invoking `OrderProductModel` method `setQuantity`. If the `Product` with the given `ISBN` is not found in the `ShoppingCart`, lines 176–178 throw a `ProductNotFoundException`.

Method `checkout` (lines 183–232) places an `Order` for the `Products` in the `ShoppingCart`. Each `Order` must have an associated `Customer`, so method `checkout` takes the `Customer`'s `userID` as an argument. Lines 192–201 create an `OrderModel` to represent the `Order`’s details. Each `Order` has an `orderDate`, a `shipped` flag and a `Collection` of `OrderProductModels`. Line 195 sets the date in the `OrderModel`. Line 198 invokes method `setOrderProductModels` of class `OrderModel` to add the `OrderProductModels` list to the `Order`. Line 201 sets the `shipped` flag to `false`, to indicate that the `Order` has not shipped from the warehouse. We discuss entity EJB `Order` in detail in Section 19.5.
Line 217 invokes method `create` of interface `OrderHome` to create a new `Order`. Method `create` takes as arguments an `OrderModel` containing the details of the `Order` to be created and a `String` containing the `Customer`'s `userID`. Line 220 empties the `ShoppingCart` by creating assigning a new `ArrayList` to `Collection` reference `orderProductModels`. Line 223 returns a remote reference to the newly created `Order`. Lines 228–230 catch any exceptions that occur.

Method `getTotal` (lines 235–257) iterates through the `Collection` of `OrderProductModel`s and calculates the total cost of items in the `ShoppingCart`.

### 19.3.3 ShoppingCartHome Interface

Interface `ShoppingCartHome` (Fig. 19.4) defines a single `create` method (lines 15–16) that creates new `ShoppingCart` EJB instances. The EJB container provides the implementation for method `create`.

Figure 19.5 and Figure 19.6 show the deployment settings for stateful session EJB `ShoppingCart`. In addition to the settings shown here, be sure to set the `Transaction Type` to `Required` for all business methods.

```java
// ShoppingCartHome.java
// ShoppingCartHome is the home interface for stateful session EJB ShoppingCart.
package com.deitel.advjhtp1.bookstore.ejb;

import java.rmi.RemoteException;

import javax.ejb.*;

public interface ShoppingCartHome extends EJBHome {

    // create new ShoppingCart EJB
    public ShoppingCart create()
        throws RemoteException, CreateException;
}
```

**Fig. 19.4** `ShoppingCartHome` interface for creating `ShoppingCart` EJB instances.

**ShoppingCart General Deployment Settings**

<table>
<thead>
<tr>
<th>Bean Type</th>
<th>Stateful Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise</td>
<td><code>com.deitel.advjhtp1.bookstore.ejb.ShoppingCartEJB</code></td>
</tr>
<tr>
<td>Bean Class</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 19.5** `ShoppingCart` general deployment settings (part 1 of 2).
Product Implementation

Entity EJB Product uses container-managed persistence to represent a Product in the Deitel Bookstore. The EJB container implements methods that select, insert, update and delete database data. The deployer must provide information about how the database table should be created and the SQL queries to be used for the create, remove and finder methods at deployment time.

19.4.1 Product Remote Interface

Remote interface Product (Fig. 19.7) declares method getProductModel (line 17–18), which returns a ProductModel that contains the Product’s details.

```java
1   // Product.java
2   // Product is the remote interface for entity EJB Product.
3   package com.deitel.advjhtpl.bookstore.ejb;
```
ProductEJB Implementation

Product remote interface implementation ProductEJB (Fig. 19.8) uses container-managed persistence. The EJB container manages the synchronization of the public data members declared on lines 18–24 with the database. Method getProductModel (lines 27–43) creates a ProductModel that contains the Product's details. Line 30 creates the ProductModel instance, and lines 33–39 invoke set methods to initialize the ProductModel's data members.

```java
// Java core packages
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;

public interface Product extends EJBObject {
    // get Product details as ProductModel
    public ProductModel getProductModel() throws RemoteException;
}
```

19.4.2 ProductEJB Implementation

Product remote interface implementation ProductEJB (Fig. 19.8) uses container-managed persistence. The EJB container manages the synchronization of the public data members declared on lines 18–24 with the database. Method getProductModel (lines 27–43) creates a ProductModel that contains the Product's details. Line 30 creates the ProductModel instance, and lines 33–39 invoke set methods to initialize the ProductModel's data members.

```java
// ProductEJB.java
// Entity EJB Product represents a Product, including the
// ISBN, publisher, author, title, price number of pages
// and cover image.
package com.deitel.advjhtp1.bookstore.ejb;

// Java extension packages
import javax.ejb.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.*;

public class ProductEJB implements EntityBean {
    private EntityContext entityContext;

    // container-managed fields
    public String ISBN;
    public String publisher;
    public String author;
    public String title;
    public double price;
```
public int pages;
public String image;

// get Product details as ProductModel
generic ProductModel getProductModel()
{
    // construct new ProductModel
    ProductModel productModel = new ProductModel();

    // initialize ProductModel with data from Product
    productModel.setISBN( ISBN );
    productModel.setPublisher( publisher );
    productModel.setAuthor( author );
    productModel.setTitle( title );
    productModel.setPrice( price );
    productModel.setPages( pages );
    productModel.setImage( image );

    return productModel;
}

// set Product details using ProductModel
private void setProductModel( ProductModel productModel )
{
    // populate Product's data members with data in
    // provided ProductModel
    publisher = productModel.getPublisher();
    author = productModel.getAuthor();
    title = productModel.getTitle();
    price = productModel.getPrice();
    pages = productModel.getPages();
    image = productModel.getImage();

} // end method setProductModel

// create instance of Product EJB using given ProductModel
generic String ejbCreate( ProductModel productModel )
{
    setProductModel( productModel );
    return null;
}

// perform any necessary post-creation tasks
public void ejbPostCreate( ProductModel productmodel ) {}
Method `setProductModel` (lines 46–58) sets the `Product`'s details, using values in the given `ProductModel`. Line 50 sets the value of the `ISBN` data member to the value of the `ISBN` contained in the `ProductModel` argument. Lines 51–56 set the values of the other `ProductEJB` data members. Method `ejbCreate` (lines 61–65) accepts a `ProductModel` argument. Method `ejbCreate` invokes method `setProductModel` with the provided `ProductModel` to initialize the `Product` EJB instance (line 63).

### 19.4.3 ProductHome Interface

Interface `ProductHome` (Fig. 19.9) creates new `ProductEJB` instances and declares `finder` methods for finding existing `Products`. Method `create` (lines 18–19) corresponds to method `ejbCreate` of Fig. 19.8, and provides an interface for creating a `ProductEJB` instance. Method `findByPrimaryKey` (lines 22–23) takes as a `String` argument the ISBN for a particular `Product` in the database. Method `findAllProducts` (lines 26–27) returns a `Collection` of all `Products` in the database. Method `findByTitle` (lines 30–31) searches for `Products` whose titles contain the given `searchString` and returns a `Collection` of `Product` remote references. The EJB container implements each `finder` method, using SQL queries the deployer must provide at deployment time.
19.4.4 ProductModel

Class **ProductModel** (Fig. 19.10) implements interface **Serializable**, so that instances may be serialized over RMI-IIOP. **ProductModel** has a **private** data member (lines 18–25) and **set** and **get** methods (lines 28–109) for each **Product** EJB property.
// third-party packages
import org.w3c.dom.*;

public class ProductModel implements Serializable, XMLGenerator {
    // ProductModel properties
    private String ISBN;
    private String publisher;
    private String author;
    private String title;
    private double price;
    private int pages;
    private String image;
    private int quantity;

    // set ISBN
    public void setISBN( String productISBN )
    {
    }

    // get ISBN
    public String getISBN()
    {
        return ISBN;
    }

    // set publisher
    public void setPublisher( String productPublisher )
    {
        publisher = productPublisher;
    }

    // get publisher
    public String getPublisher()
    {
        return publisher;
    }

    // set author
    public void setAuthor( String productAuthor )
    {
        author = productAuthor;
    }

    // get author
    public String getAuthor()
    {
        return author;
    }
}

Fig. 19.10 ProductModel class for serializing Product data (part 2 of 4).
public void setTitle( String productTitle )
{
    title = productTitle;
}

public String getTitle()
{
    return title;
}

public void setPrice( double productPrice )
{
    price = productPrice;
}

public double getPrice()
{
    return price;
}

public void setPages( int pageCount )
{
    pages = pageCount;
}

public int getPages()
{
    return pages;
}

public void setImage( String productImage )
{
    image = productImage;
}

public String getImage()
{
    return image;
}

public Element getXML( Document document )
{
    // create product Element
    Element product = document.createElement( "product" );

    // set title
    product.appendChild( document.createTextNode( title ) );

    // set price
    product.appendChild( document.createTextNode( price ) );

    // set number of pages
    product.appendChild( document.createTextNode( pages ) );

    // set URL of cover image
    product.appendChild( document.createTextNode( image ) );

    return product;
}
ProductModel also implements interface XMLGenerator (Fig. 19.11), which defines a single method, getXML. Method getXML of class ProductModel (lines 112–
generates an XML Element for the data contained in the ProductModel. This method uses the Document argument to create XML elements for each ProductModel data member. However, method getXML does not modify the Document. Line 162 returns the newly created product element.

Figure 19.12 and Fig. 19.13 show the deployment settings for entity EJB Product. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods.

```java
// XMLGenerator.java
// XMLGenerator is an interface for classes that can generate XML Elements. The XML element returned by method getXML should contain Elements for each public property.
package com.deitel.advjhtp1.bookstore.model;

// third-party packages
import org.w3c.dom.*;

public interface XMLGenerator {
    // build an XML element for this Object
    public Element getXML( Document document );
}
```

Fig. 19.11 XMLGenerator interface for generating XML Elements for public properties.

### Product General Deployment Settings

<table>
<thead>
<tr>
<th>Bean Type</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise Bean Class</td>
<td>com.deitel.advjhtp1.bookstore.ejb.ProductEJB</td>
</tr>
<tr>
<td>Home Interface</td>
<td>com.deitel.advjhtp1.bookstore.ejb.ProductHome</td>
</tr>
<tr>
<td>Remote Interface</td>
<td>com.deitel.advjhtp1.bookstore.ejb.Product</td>
</tr>
</tbody>
</table>

Fig. 19.12 Product general deployment settings.

### Product Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Persistence Management</th>
<th>Container-Managed Persistence</th>
</tr>
</thead>
</table>

Fig. 19.13 Product Entity and deployment settings (part 1 of 2).
### Product Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Primary Key Class</th>
<th>java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key Field Name</td>
<td>ISBN</td>
</tr>
<tr>
<td>Database JNDI Name</td>
<td>jdbc/Bookstore</td>
</tr>
<tr>
<td>Method findByTitle SQL Statement</td>
<td>SELECT ISBN FROM Product WHERE title LIKE ?1</td>
</tr>
<tr>
<td>Method findAllProducts SQL Statement</td>
<td>SELECT ISBN FROM Product WHERE 1 = 1</td>
</tr>
<tr>
<td>Method ejbRemove SQL Statement</td>
<td>DELETE FROM Product WHERE ISBN = ?</td>
</tr>
<tr>
<td>Method findByPrimaryKey SQL Statement</td>
<td>SELECT ISBN FROM Product WHERE ISBN = ?</td>
</tr>
<tr>
<td>Method ejbLoad SQL Statement</td>
<td>SELECT author, image, pages, price, publisher, title FROM Product WHERE ISBN = ?</td>
</tr>
<tr>
<td>Table Create SQL Statement</td>
<td>CREATE TABLE Product ( ISBN VARCHAR(255), author VARCHAR(255), image VARCHAR(255), pages INTEGER NOT NULL, price DOUBLE PRECISION NOT NULL, publisher VARCHAR(255), title VARCHAR(255), CONSTRAINT pk_Product PRIMARY KEY ( ISBN ) )</td>
</tr>
<tr>
<td>Table Delete SQL Statement</td>
<td>DROP TABLE Product</td>
</tr>
</tbody>
</table>

Fig. 19.13 Product Entity and deployment settings (part 2 of 2).

### 19.5 Order Implementation

Entity EJB Order represents an Order placed at the Deitel Bookstore. Each Order consists of a list of Products and their associated quantities, as well as the customerId of the Customer who placed the Order.
19.5.1 Order Remote Interface

The Order EJB remote interface (Fig. 19.14) defines the business methods available in the Order EJB. Method getOrderModel (line 17) returns an OrderModel containing Order details. Method setShipped (lines 20–21) marks an Order as having been shipped from the warehouse. Method isShipped (line 24) returns a boolean that indicates whether the Order has been shipped.

19.5.2 OrderEJB Implementation

Order remote interface implementation OrderEJB (Fig. 19.15) declares public data members for container-manage persistence (lines 26–29). Method getOrderModel (lines 32–91) constructs an OrderModel instance that contains Order details. Lines 42–44 populate the OrderModel with the values of the Order EJB’s data members.

```java
// Order.java
// Order is the remote interface for entity EJB Order.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;

public interface Order extends EJBObject {
    // get Order details as OrderModel
    public OrderModel getOrderModel() throws RemoteException;

    // set shipped flag
    public void setShipped( boolean flag )
        throws RemoteException;

    // get shipped flag
    public boolean isShipped() throws RemoteException;
}
```

Fig. 19.14 Order remote interface for modifying details of Order EJB instances.

```java
// OrderEJB.java
// Entity EJB Order represents an Order, including the
// orderID, Order date, total cost and whether the Order
// has shipped.
package com.deitel.advjhtp1.bookstore.ejb;
```

Fig. 19.15 OrderEJB implementation of Order remote interface (part 1 of 6).
```java
public class OrderEJB implements EntityBean {
    private EntityContext entityContext;
    private InitialContext initialContext;
    private DateFormat dateFormat;

    // container-managed fields
    public Integer orderID;
    public Integer customerID;
    public String orderDate;
    public boolean shipped;

    // get Order details as OrderModel
    public OrderModel getOrderModel() throws EJBException {
        // construct new OrderModel
        OrderModel orderModel = new OrderModel();

        // look up OrderProduct EJB to retrieve list
        // of Products contained in the Order
        try {
            initialContext = new InitialContext();
            Object object = initialContext.lookup(
                    "java:comp/env/ejb/OrderProduct");
            OrderProductHome orderProductHome =
                (OrderProductHome) PortableRemoteObject.narrow(
                    object, OrderProductHome.class);
            Collection orderProducts =
                orderProductHome.findByOrderID( orderID );

            Iterator iterator = orderProducts.iterator();
```
// OrderProductModels to place in OrderModel
Collection orderProductModels = new ArrayList();

// get OrderProductModel for each Product in Order
while ( iterator.hasNext() ) {
    OrderProduct orderProduct = ( OrderProduct )
        PortableRemoteObject.narrow( iterator.next(),
            OrderProduct.class );

    // get OrderProductModel for OrderProduct record
    OrderProductModel orderProductModel =
        orderProduct.getOrderProductModel();

    // add OrderProductModel to list of
    // OrderProductModels in the Order
    orderProductModels.add( orderProductModel );
}

// add Collection of OrderProductModels to OrderModel
orderModel.setOrderProductModels( orderProductModels );

} // end try

// handle exception working with OrderProduct EJB
catch ( Exception exception ) {
    throw new EJBException( exception );
}

return orderModel;

} // end method getOrderModel

// set shipped flag
public void setShipped( boolean flag )
{
    shipped = flag;
}

// get shipped flag
public boolean isShipped()
{
    return shipped;
}

// create new Order EJB using given OrderModel and userID
public Integer ejbCreate( OrderModel order, String userID )
throws CreateException
{
    // retrieve unique value for primary key of this
    // Order using SequenceFactory EJB
    try {
        initialContext = new InitialContext();

Fig. 19.15 OrderEJB implementation of Order remote interface (part 3 of 6).
Object object = initialContext.lookup("java:comp/env/ejb/SequenceFactory");

SequenceFactoryHome sequenceFactoryHome = 
   (SequenceFactoryHome)
   PortableRemoteObject.narrow(
       object, SequenceFactoryHome.class);

// find sequence for CustomerOrder table
SequenceFactory sequenceFactory =
   sequenceFactoryHome.findByPrimaryKey("CustomerOrders");

// get next unique orderID
orderID = sequenceFactory.getNextID();

// get date, cost, shipped flag and list of
// OrderProduct from provided OrderModel
orderDate = dateFormat.format(order.getOrderDate());
shipped = order.getShipped();

// get OrderProductModels that comprise OrderModel
Collection orderProductModels =
   order.getOrderProductModels();

// create OrderProduct EJBs for each Product in
// Order to keep track of quantity
object = initialContext.lookup("java:comp/env/ejb/OrderProduct");

OrderProductHome orderProductHome =
   (OrderProductHome)
   PortableRemoteObject.narrow(
       object, OrderProductHome.class);

Iterator iterator = orderProductModels.iterator();

// create an OrderProduct EJB with Product's
// ISBN, quantity and orderID for this Order
while (iterator.hasNext()) {

    OrderProductModel orderProductModel =
        (OrderProductModel) iterator.next();

    // set orderID for OrderProduct record
    orderProductModel.setOrderID(orderID);

    // create OrderProduct EJB instance
    orderProductHome.create(orderProductModel);
}

// get customerID for customer placing Order
object = initialContext.lookup(Fig. 19.15 OrderEJB implementation of Order remote interface (part 4 of 6).
"java:comp/env.ejb/Customer" );

CustomerHome customerHome =
    ( CustomerHome ) PortableRemoteObject.narrow(
        object, CustomerHome.class );

    // use provided userID to find Customer
    Customer customer =
        customerHome.findByUserID( userID );
    
        customerId = ( Integer ) customer.getPrimaryKey();

    } // end try

    // handle exception when looking up EJBs
    catch ( Exception exception ) {
        throw new CreateException( exception.getMessage() );
    }

    return null;

} // end method ejbCreate

    // perform any necessary post-creation tasks
    public void ejbPostCreate( OrderModel order, String id ) {

    }

    // set EntityContext
    public void setEntityContext( EntityContext context )
    {
        entityContext = context;
        dateFormat = DateFormat.getDateTimeInstance(
            DateFormat.FULL, DateFormat.SHORT, Locale.US );
    }

    // unset EntityContext
    public void unsetEntityContext()
    {
        entityContext = null;
    }

    // activate Order EJB instance
    public void ejbActivate()
    {
        orderID = ( Integer ) entityContext.getPrimaryKey();
    }

    // passivate Order EJB instance
    public void ejbPassivate()
    {
        orderID = null;
    }

    // remove Order EJB instance

Fig. 19.15 OrderEJB implementation of Order remote interface (part 5 of 6).
In addition to the `orderDate`, `orderID` and `shipped` flag, an `Order` contains a list of `OrderProductModels`. The relationship between an `Order` and its associated `Products` and quantities is represented by a many-to-many relationship in the database (i.e., an `Order` can contain many `Products`, and a single `Product` can be found in many `Orders`). The `OrderProduct` EJB represents this relationship by mapping an `orderID` to the ISBNs of `Products` in the `Order`. For each `Order`, there are records in the `OrderProduct` table containing the ISBNs and quantities of each `Product` in the `Order`. For example, if a `Customer` orders one copy of `Java How to Program` and two copies of `Advanced Java 2 Platform How to Program`, there will be two records in the `OrderProduct` table. Each record will have the same `orderID`, but one will have the ISBN for `Java How to Program` and the `quantity1`, and the other will have the ISBN for `Advanced Java 2 Platform How to Program` and the `quantity2`.

Lines 56–57 invoke method `findByOrderID` to obtain the `OrderProduct` records for the `Order`. Method `findByOrderID` returns a `Collection` of `OrderProduct` remote references. Lines 65–77 traverse the `Collection` of `OrderProducts` using an `Iterator`, and lines 71–72 retrieve an `OrderProductModel` for each `OrderProduct` record. Line 76 adds each `OrderProductModel` to a `Collection`. Line 80 adds the `OrderProductModelCollection` to the `OrderModel`, and line 89 returns the newly created `OrderModel`.

Method `setShipped` (lines 94–97) accepts a `boolean` argument and updates the `Order` EJB’s `shipped` flag. An order-tracking application could use method `setShipped` to update the status of the `Order` when the warehouse ships the `Order`. Method `isShipped` (lines 100–103) returns the `shipped` data member’s current value, indicating whether the `Order` has been shipped from the warehouse.

Method `ejbCreate` (lines 106–187) creates an `Order` EJB using data from the given `OrderModel` and `userID`. Each `Order` has an associated `orderID`, which serves as the primary key in the `Order` EJB database table. `SequenceFactory` method `getNextID` generates a unique `orderID` for the `Order` (line 128).

Lines 132–133 populate the `Order` with data from the `OrderModel`. The `OrderModel` also provides the `Products` and quantities in the `Order` as a `Collection` of `OrderProductModels`. Lines 152–162 process the `Collection` of `OrderProductModels` and create `OrderProduct` records for each, using interface `OrderProductHome` method `create` (line 161).

Each `Order` also must have an associated `Customer` who placed the `Order`. This is a one-to-many relationship, because one `Customer` can place many orders, but an `Order` can be associated with only one `Customer`. Lines 173–174 retrieve the `Cus-
Customer EJB for the given userID. Customer method getPrimaryKey retrieves the Customer’s customerID. Line 176 sets the customerID for the Order.

### 19.5.3 OrderHome Interface

Interface OrderHome (Fig. 19.16) creates Order instances and finds existing Orders. Method create (lines 18–19) corresponds to method ejbCreate of Fig. 19.15 and creates new Orders, using an OrderModel and userID. Method findByPrimaryKey (lines 22–23) locates an existing Order, using its orderID. Method findByCustomerID (lines 26–27) retrieves a Collection of Orders for the given Customer.

### 19.5.4 OrderModel

Class OrderModel (Fig. 19.17) encapsulates the details of an Order EJB in a Serializable object suitable for delivery over RMI-IIOP. OrderModel has private data members (lines 19–22) with associated set and get methods (lines 31–102) for each Order EJB data member. OrderModel also maintains a Collection of OrderProductModels, to keep track of the Products in the Order. OrderModel implements interface XMLGenerator and method getXML to facilitate the generation of an XML representation of an Order (lines 105–166).

```java
// OrderHome.java
// OrderHome is the home interface for entity EJB Order.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.util.*;
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model._CITY;

public interface OrderHome extends EJBHome {
    // create Order using given OrderModel and userID
    public Order create( OrderModel orderModel, String userID )
        throws RemoteException, CreateException;

    // find Order using given orderID
    public Order findByPrimaryKey( Integer orderID )
        throws RemoteException, FinderException;

    // find Orders for given customerID
    public Collection findByCustomerID( Integer customerID )
        throws RemoteException, FinderException;
}
```

Fig. 19.16 OrderHome interface for finding and creating Order EJB instances.
// OrderModel.java
// OrderModel represents an Order and contains the order ID,
// date, total cost and a boolean indicating whether or not the
// order has shipped.
package com.deitel.advjhtp1.bookstore.model;

// Java core packages
import java.io.*;
import java.util.*;
import java.text.*;

// third-party packages
import org.w3c.dom.*;

public class OrderModel implements Serializable, XMLGenerator {

    // OrderModel properties
    private Integer orderID;
    private Date orderDate;
    private boolean shipped;
    private Collection orderProductModels;

    // construct empty OrderModel
    public OrderModel() {
        orderProductModels = new ArrayList();
    }

    // set order ID
    public void setOrderID( Integer id ) {
        orderID = id;
    }

    // get order ID
    public Integer getOrderID() {
        return orderID;
    }

    // set order date
    public void setOrderDate( Date date ) {
        orderDate = date;
    }

    // get order date
    public Date getOrderDate() {
        return orderDate;
    }

Fig. 19.17 OrderModel class for serializing Order data (part 1 of 4).
// get total cost
public double getTotalCost()
{
    double total = 0.0;

    Iterator iterator = orderProductModels.iterator();

    // calculate Order's total cost
    while ( iterator.hasNext() ) {
        // get next OrderProduct in ShoppingCart
        OrderProductModel orderProductModel =
            ( OrderProductModel ) iterator.next();

        ProductModel productModel =
            orderProductModel.getProductModel();

        // add OrderProduct extended price to total
        total += ( productModel.getPrice() * 
            orderProductModel.getQuantity() );
    }

    return total;
}

// set shipped flag
public void setShipped( boolean orderShipped )
{
    shipped = orderShipped;
}

// get shipped flag
public boolean getShipped()
{
    return shipped;
}

// set list of OrderProductModels
public void setOrderProductModels( Collection models )
{
    orderProductModels = models;
}

// get OrderProductModels
public Collection getOrderProductModels()
{
    return Collections.unmodifiableCollection(
        orderProductModels );
}

// get XML representation of Order
public Element getXML( Document document )
{
Element order = document.createElement( "order" );

Element temp = document.createElement( "orderID" );
temp.appendChild( document.createTextNode( String.valueOf( getOrderID() ) ) );
order.appendChild( temp );

// get DateFormat for writing Date to XML document
DateFormat formatter = DateFormat.getDateTimeInstance( DateFormat.DEFAULT, DateFormat.MEDIUM, Locale.US );

// create orderDate Element
temp = document.createElement( "orderDate" );
temp.appendChild( document.createTextNode( formatter.format( getOrderDate() ) ) );
order.appendChild( temp );

NumberFormat costFormatter = NumberFormat.getCurrencyInstance( Locale.US );

// create totalCost Element
temp = document.createElement( "totalCost" );
temp.appendChild( document.createTextNode( costFormatter.format( getTotalCost() ) ) );
order.appendChild( temp );

// create shipped Element
temp = document.createElement( "shipped" );
if ( getShipped() )
    temp.appendChild( document.createTextNode( "yes" ) );
else
    temp.appendChild( document.createTextNode( "no" ) );
order.appendChild( temp );

// create orderProducts Element
Element orderProducts = document.createElement( "orderProducts" );

Iterator iterator = getOrderProductModels().iterator();

// add orderProduct element for each OrderProduct
while ( iterator.hasNext() ) {
    OrderProductModel orderProductModel = ( OrderProductModel ) iterator.next();
    orderProducts.appendChild( orderProductModel.getXML( document ) );

Fig. 19.17 OrderModel class for serializing Order data (part 3 of 4).
Figure 19.18, Fig. 19.19 and Fig. 19.20 show the deployment settings for entity EJB Order. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods.

Fig. 19.17 OrderModel class for serializing Order data (part 4 of 4).

Figure 19.18, Fig. 19.19 and Fig. 19.20 show the deployment settings for entity EJB Order. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods.

**Order General Deployment Settings**

<table>
<thead>
<tr>
<th>Bean Type</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise</td>
<td>com.deitel.advjhtp1.bookstore.ejb.OrderEJB</td>
</tr>
<tr>
<td>Bean Class</td>
<td>com.deitel.advjhtp1.bookstore.ejb.OrderEJB</td>
</tr>
<tr>
<td>Home Interface</td>
<td>com.deitel.advjhtp1.bookstore.ejb.OrderHome</td>
</tr>
<tr>
<td>Remote Interface</td>
<td>com.deitel.advjhtp1.bookstore.ejb.Order</td>
</tr>
</tbody>
</table>

**Order Entity and Deployment Settings**

<table>
<thead>
<tr>
<th>Persistence Management</th>
<th>Container-Managed Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key Class</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>Primary Key Field Name</td>
<td>orderID</td>
</tr>
<tr>
<td>Database JNDI Name</td>
<td>jdbc/Bookstore</td>
</tr>
<tr>
<td>Method</td>
<td>findBy-CustomerID</td>
</tr>
<tr>
<td>SQL Statement</td>
<td>SELECT orderID FROM CustomerOrders WHERE customerID = ?1</td>
</tr>
</tbody>
</table>

**Fig. 19.18 Order** general deployment settings.

**Fig. 19.19 Order** entity and deployment settings (part 1 of 2).
### Order Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Method</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejb-Store</td>
<td>UPDATE CustomerOrders SET customerID = ?, orderDate = ?, shipped = ? WHERE orderID = ?</td>
</tr>
<tr>
<td>ejbCreate</td>
<td>INSERT INTO CustomerOrders ( customerID, orderDate, orderID, shipped ) VALUES ( ?, ?, ?, ? )</td>
</tr>
<tr>
<td>ejbRemove</td>
<td>DELETE FROM CustomerOrders WHERE orderID = ?</td>
</tr>
<tr>
<td>findByPrimaryKey</td>
<td>SELECT orderID FROM CustomerOrders WHERE orderID = ?</td>
</tr>
<tr>
<td>ejbLoad</td>
<td>SELECT customerID, orderDate, shipped FROM CustomerOrders WHERE orderID = ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Create</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE</td>
<td>CustomerOrders ( customerID INTEGER, orderDate VARCHAR(255), orderID INTEGER, shipped BOOLEAN NOT NULL, CONSTRAINT pk_CustomerOrders PRIMARY KEY ( orderID ) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Delete</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP TABLE</td>
<td>CustomerOrders</td>
</tr>
</tbody>
</table>

**Fig. 19.19** Order entity and deployment settings (part 2 of 2).

### Order EJB References

<table>
<thead>
<tr>
<th>Coded Name</th>
<th>ejb/Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Entity</td>
</tr>
<tr>
<td>Home</td>
<td>com.deitel.advjhtp1.bookstore.ejb.ProductHome</td>
</tr>
<tr>
<td>Remote</td>
<td>com.deitel.advjhtp1.bookstore.ejb.Product</td>
</tr>
<tr>
<td>JNDI Name</td>
<td>Product</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coded Name</th>
<th>ejb/SequenceFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Entity</td>
</tr>
<tr>
<td>Home</td>
<td>com.deitel.advjhtp1.bookstore.ejb.SequenceFactory-Home</td>
</tr>
<tr>
<td>Remote</td>
<td>com.deitel.advjhtp1.bookstore.ejb.SequenceFactory</td>
</tr>
<tr>
<td>JNDI Name</td>
<td>SequenceFactory</td>
</tr>
</tbody>
</table>

| Coded Name       | ejb/Customer                                                                |

**Fig. 19.20** Order EJB references (part 1 of 2).
19.6 OrderProduct Implementation

Entity EJB OrderProduct represents the many-to-many relationship between Orders and Products. Each OrderProduct EJB includes an orderID, Product ISBN and quantity, and represents one line item in an Order.

19.6.1 OrderProduct Remote Interface

The OrderProduct remote interface (Fig. 19.21) defines the business methods for the OrderProduct EJB. The OrderProduct EJB maps Product ISBNs to orderIDs and quantities. Method getOrderProductModel (lines 18–19) returns an OrderProductModel that contains the details of an OrderProduct record.

```java
// OrderProduct.java
// OrderProduct is the remote interface for entity EJB
// OrderProduct.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;

public interface OrderProduct extends EJBObject {
  // get OrderProduct details as OrderProductModel
```

Fig. 19.21 OrderProduct remote interface for modifying details of OrderProduct EJB instances (part 1 of 2).
**19.6.2 OrderProductEJB Implementation**

OrderProduct remote interface implementation OrderProductEJB (Fig. 19.22) declares container-managed fields ISBN, orderID and quantity (lines 22–24). Method getOrderProductModel (lines 27–69) returns the details of the OrderProduct record as an OrderProductModel. Method setOrderProductModel (lines 72–77) sets the details of the OrderProduct record, using data from the OrderProductModel argument.

```java
// OrderProductEJB.java
// Entity EJB OrderProductEJB represents the mapping between
// a Product and an Order, including the quantity of the
// Product in the Order.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;

public class OrderProductEJB implements EntityBean {
    private EntityContext entityContext;

    // container-managed fields
    public String ISBN;
    public Integer orderID;
    public int quantity;

    // get OrderProduct details as OrderProductModel
    public OrderProductModel getOperationModel() throws EJBException
    {
        OrderProductModel model = new OrderProductModel();

        // get ProductModel for Product in this OrderProduct
        try {
            Context initialContext = new InitialContext();
        }
    }
}
```

Fig. 19.21 OrderProduct remote interface for modifying details of OrderProduct EJB instances (part 2 of 2).
// look up Product EJB
Object object = initialContext.lookup(
    "java:comp/env.ejb/Product" );

// get ProductHome interface
ProductHome productHome = ( ProductHome )
    PortableRemoteObject.narrow( object,
    ProductHome.class );

// find Product using its ISBN
Product product =
    productHome.findByPrimaryKey( ISBN );

// get ProductModel
ProductModel productModel =
    product.getProductModel();

// set ProductModel in OrderProductModel
model.setProductModel( productModel );

} // end try

// handle exception when looking up Product EJB
catch ( Exception exception ) {
    throw new EJBException( exception );
}

// set orderID and quantity in OrderProductModel
model.setOrderID( orderID );
model.setQuantity( quantity );

    return model;

} // end method getOrderProductModel

// set OrderProduct details using OrderProductModel
private void setOrderProductModel( OrderProductModel model )
{
    orderID = model.getOrderID();
    quantity = model.getQuantity();
}

// create OrderProduct for given OrderProductModel
public OrderProductPK ejbCreate( OrderProductModel model )
{
    setOrderProductModel( model );
    return null;
}

} // perform any necessary post-creation tasks

Fig. 19.22 OrderProductEJB implementation of OrderProduct remote interface (part 2 of 3).
The EJB container calls method `ejbCreate` (lines 80–84) to create new instances of the `OrderProductEJB` EJB. The ISBN, orderID and quantity for the `OrderProduct` record are provided in the `OrderProductModel` argument. Line 82 invokes method `setOrderProductModel` to complete the creation of the `OrderProduct` record.

### 19.6.3 OrderProductHome Interface

Interface `OrderProductHome` (Fig. 19.23) provides methods for creating new `OrderProduct` EJB instances and locating existing `OrderProduct` records. Method `create` (lines 19–20) corresponds to method `ejbCreate` of the `OrderProductEJB` implementation.
tation (Fig. 19.22). Method findByOrderID (lines 23–24) locates all OrderProduct records with the provided orderID and returns a Collection of OrderProduct remote references. Method findByPrimaryKey (lines 27–28) locates an OrderProduct record, using an instance of the OrderProductPK primary-key class.

19.6.4 OrderProductPK Primary-Key Class

OrderProductPK (Fig. 19.24) is the primary-key class for the OrderProduct EJB. An orderID and ISBN are required to identify a particular OrderProduct EJB instance uniquely. An entity EJB that has a complex primary key (i.e., a primary key that consists of multiple fields) requires a custom primary-key class. A custom primary-key class must have a public data member for each field in the complex primary key. The primary-key class OrderProductPK has two public data members (lines 12–13)—ISBN and orderID, which correspond to OrderProductEJB’s two primary-key fields (Fig. 19.22). A custom primary-key class also must override methods hashCode and equals of class Object. The overridden implementations of methods hashCode (lines 38–41) and equals (lines 44–57) enable the EJB container and OrderProduct EJB clients to determine if two OrderProduct EJB instances are equal by comparing their primary-key class instances.

```java
// OrderProductHome.java
// OrderProductHome is the home interface for entity EJB
// OrderProduct.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.util.Collection;
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;

public interface OrderProductHome extends EJBHome {
    // create OrderProduct using given OrderProductModel
    public OrderProduct create( OrderProductModel model )
        throws RemoteException, CreateException;

    // find OrderProduct for given orderID
    public Collection findByOrderID( Integer orderID )
        throws RemoteException, FinderException;

    // find OrderProduct for given primary key
    public OrderProduct findByPrimaryKey( OrderProductPK pk )
        throws RemoteException, FinderException;
}
```

Fig. 19.23 OrderProductHome interface for finding and creating OrderProduct EJB instances.
// OrderProductPK.java
// OrderProductPK is a primary-key class for entity EJB
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.io.*;

public class OrderProductPK implements Serializable {

    // primary-key fields
    public String ISBN;
    public Integer orderID;

    // no-argument constructor
    public OrderProductPK() {}

    // construct OrderProductPK with ISBN and orderID
    public OrderProductPK(String isbn, Integer id) {
        ISBN = isbn;
        orderID = id;
    }

    // get ISBN
    public String getISBN() {
        return ISBN;
    }

    // get orderID
    public Integer getOrderID() {
        return orderID;
    }

    // calculate hashCode for this Object
    public int hashCode() {
        return getISBN().hashCode() ^ getOrderID().intValue();
    }

    // custom implementation of Object equals method
    public boolean equals(Object object) {
        // ensure object is instance of OrderProductPK
        if (object instanceof OrderProductPK) {
            OrderProductPK otherKey =
                (OrderProductPK) object;

            // compare ISBNs and orderIDs
            return (getISBN().equals(otherKey.getISBN())

    Fig. 19.24 OrderProductPK primary-key class for OrderProduct EJB (part 1 of 2).
19.6.5 OrderProductModel

Model class OrderProductModel (Fig. 19.25) represents an OrderProduct record. Line 18 declares a ProductModel reference for the Product associated with this OrderProduct record. The quantity (line 19) is the quantity of the Product in the Order. The Order is identified by its orderID (line 20). Method getXML (lines 59–76) generates an XML Element that represents the OrderProductModel.

```java
// OrderProductModel.java
// OrderProductModel represents a Product and its quantity in an Order or ShoppingCart.
package com.deitel.advjhtp1.bookstore.model;

// Java core packages
import java.io.*;
import java.util.*;
import java.text.*;

// third-party packages
import org.w3c.dom.*;

public class OrderProductModel implements Serializable, XMLGenerator {

    // OrderProductModel properties
    private ProductModel productModel;
    private int quantity;
    private Integer orderID;

    // set ProductModel
    public void setProductModel( ProductModel model )
    {
        productModel = model;
    }

    // get ProductModel
    public ProductModel getProductModel()
    {
        return productModel;
    }
}
```

Fig. 19.24 OrderProductPK primary-key class for OrderProduct EJB (part 2 of 2).

Fig. 19.25 OrderProductModel class for serializing OrderProduct data (part 1 of 2).
Figure 19.25  

OrderProductModel class for serializing OrderProduct data (part 2 of 2).

Figure 19.26, Fig. 19.27 and Fig. 19.28 show the deployment settings for entity EJB OrderProduct. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods. Note that because entity EJB OrderProduct uses a custom primary-key class, the Primary Key Field Name (Fig. 19.27) must be left blank.
### OrderProduct General Deployment Settings

<table>
<thead>
<tr>
<th>Bean Type</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise Bean Class</td>
<td>com.deitel.advjhtpl.bookstore.ejb.OrderProductEJB</td>
</tr>
<tr>
<td>Home Interface</td>
<td>com.deitel.advjhtpl.bookstore.ejb.OrderProductHome</td>
</tr>
<tr>
<td>Remote Interface</td>
<td>com.deitel.advjhtpl.bookstore.ejb.OrderProduct</td>
</tr>
</tbody>
</table>

**Fig. 19.26** *OrderProduct* general deployment settings.

### OrderProduct Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Persistence Management</th>
<th>Container-Managed Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key Class</td>
<td>com.deitel.advjhtpl.bookstore.ejb.OrderProductPK</td>
</tr>
<tr>
<td>Primary Key Field Name</td>
<td>N/A</td>
</tr>
<tr>
<td>Database JNDI Name</td>
<td>jdbc/Bookstore</td>
</tr>
<tr>
<td>Method findByOrderID</td>
<td>SELECT ISBN, orderID FROM OrderProduct WHERE orderID = ?1</td>
</tr>
<tr>
<td>SQL Statement</td>
<td></td>
</tr>
<tr>
<td>SQL Statement</td>
<td></td>
</tr>
<tr>
<td>SQL Statement</td>
<td></td>
</tr>
<tr>
<td>SQL Statement</td>
<td></td>
</tr>
<tr>
<td>Method findByPrimaryKey</td>
<td>SELECT ISBN, orderID FROM OrderProduct WHERE ISBN = ? AND orderID = ?</td>
</tr>
<tr>
<td>SQL Statement</td>
<td></td>
</tr>
<tr>
<td>Method ejbLoad</td>
<td>SELECT quantity FROM OrderProduct WHERE ISBN = ? AND orderID = ?</td>
</tr>
<tr>
<td>SQL Statement</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 19.27** *OrderProduct* entity and deployment settings (part 1 of 2).
In this chapter, we presented the business logic for managing the Customer’s ShoppingCart and the data model for Products and Orders in our on-line store. We also discussed how Serializable objects can be used to reduce network traffic when communicating with EJBs. In the next chapter, we present EJBs for managing Customers of the on-line store.

| Table Create SQL Statement | CREATE TABLE OrderProduct ( ISBN VARCHAR(255), orderID INTEGER, quantity INTEGER NOT NULL, CONSTRAINT pk_OrderProduct PRIMARY KEY ( ISBN, orderID ) ) |
| Table Delete SQL Statement | DROP TABLE OrderProduct |

**Fig. 19.27** OrderProduct entity and deployment settings (part 2 of 2).

**OrderProduct EJB References**

| Coded Name | ejb/Product |
| Type       | Entity      |
| Home       | com.deitel.advjhtp1.bookstore.ejb.ProductHome |
| Remote     | com.deitel.advjhtp1.bookstore.ejb.Product |
| JNDI Name  | Product     |

**Fig. 19.28** OrderProduct EJB references.
Objectives

- To understand the data model for Customer management in the Deitel Bookstore case study.
- To implement an EJB for storing billing and shipping information.
- To build an EJB for generating primary keys.
- To understand the benefits of declarative transaction semantics.
- To understand the steps necessary to deploy the Deitel Bookstore case study.

*The best investment is in the tools of one’s own trade.*
Benjamin Franklin

*Creativity is not the finding of a thing, but the making something out of it after it is found.*
James Russell Lowell

*Events that are predestined require but little management. They manage themselves.*
Amelia Barr

*All meanings, we know, depend on the key of interpretation.*
George Eliot
20.1 Introduction

In this chapter, we present entity EJBs for managing customers. Maintaining information about the customers of an on-line store can make purchases more convenient, because billing and shipping information is stored on the server. The on-line store’s marketing department also can use gathered data for distribution of marketing materials and analysis of demographic information.

We also present an entity EJB that generates unique IDs for the Customer, Order and Address EJBs. Instances of these EJBs are created when new Customers register and when Customers place new Orders. Relational databases require unique primary keys to maintain referential integrity and perform queries. We provide the Sequence-Factory EJB to generate these unique IDs, because not all databases can generate these primary-key values automatically. Finally, we provide instructions for deploying the Deitel Bookstore case study on Sun Microsystems’ J2EE reference implementation application server.

20.2 Customer Implementation

The Customer entity EJB represents a customer in the underlying database. The following subsections present the EJB implementation and its corresponding model class.
20.2.1 Customer Remote Interface

The Customer EJB remote interface (Fig. 20.1) defines the business logic methods in the EJB. Method getCustomerModel (lines 19–20) builds a Serializable object of class CustomerModel, which contains the details of the Customer. Method getOrderHistory (lines 23–24) returns a Collection of OrderModels that contains information about past Orders the Customer has placed. Method getPasswordHint (line 27) returns a String containing a hint to remind Customers of forgotten passwords.

20.2.2 CustomerEJB Implementation

The CustomerEJB implementation (Fig. 20.2) of remote interface Customer (Fig. 20.1) contains instance variables for each Customer property (lines 25–36). These instance variables are public, so the EJB container can synchronize their values with the associated database table.

```java
// Customer.java
// Customer is the remote interface for entity EJB Customer.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core libraries
import java.rmi.RemoteException;
import java.util.ArrayList;

// Java standard extensions
import javax.ejb.*;

// Deitel Bookstore libraries
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public interface Customer extends EJBObject {
    // get Customer data as a CustomerModel
    public CustomerModel getCustomerModel() throws RemoteException;

    // get Order history for CustomerModel
    public Collection getOrderHistory() throws RemoteException, NoOrderHistoryException;

    // get password hint for CustomerModel
    public String getPasswordHint() throws RemoteException;
}
```

Fig. 20.1 Customer remote interface for modifying Customer details, getting an Order history and password hint.
// CustomerEJB.java
// Entity EJB Customer represents a Customer, including
// the Customer's user name, password, billing
// address, shipping address and credit card information.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.util.*;
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public class CustomerEJB implements EntityBean {
    private EntityContext entityContext;
    private InitialContext initialContext;

    // container-managed fields
    public Integer customerID;
    public String userID;
    public String password;
    public String passwordHint;
    public String firstName;
    public String lastName;
    public Integer billingAddressID;
    public Integer shippingAddressID;

    public String creditCardName;
    public String creditCardNumber;
    public String creditCardExpirationDate;

    // get CustomerModel
    public CustomerModel getCustomerModel() throws EJBException {
        // construct new CustomerModel
        CustomerModel customer = new CustomerModel();

        // populate CustomerModel with data for this Customer
        customer.setCustomerID( customerID );
        customer.setUserID( userID );
        customer.setPassword( password );
        customer.setPasswordHint( passwordHint );
        customer.setFirstName( firstName );
        customer.setLastName( lastName );
        customer.setBillingAddressID( billingAddressID );
        customer.setShippingAddressID( shippingAddressID );

        // use Address EJB to get Customer billing and shipping

Fig. 20.2 CustomerEJB implementation of Customer remote interface (part 1 of 6).
try {
    initialContext = new InitialContext();
    Object object = initialContext.lookup("java:comp/env/ejb/Address");

    AddressHome addressHome = (AddressHome)
                        PortableRemoteObject.narrow(object, AddressHome.class);

    // get remote reference to billing Address
    Address billingAddress =
                                addressHome.findByPrimaryKey(billingAddressID);

    // add billing AddressModel to CustomerModel
    customer.setBillingAddress(billingAddress.getAddressModel());

    // get remote reference to shipping Address
    Address shippingAddress =
                                addressHome.findByPrimaryKey(shippingAddressID);

    // add shipping AddressModel to CustomerModel
    customer.setShippingAddress(shippingAddress.getAddressModel());

} // end try

// handle exception using Address EJB
catch (Exception exception) {
  throw new EJBException(exception);
}

// set credit card information in CustomerModel
customer.setCreditCardName(creditCardName);
customer.setCreditCardNumber(creditCardNumber);
customer.setCreditCardExpirationDate(creditCardExpirationDate);

return customer;

} // end method getCustomerModel

// get Order history for Customer
public Collection getOrderHistory() throws NoOrderHistoryException, EJBException {

  Collection history = new ArrayList();

  // use Order EJB to obtain list of Orders for Customer
  try {
    } // end try

Fig. 20.2 CustomerEJB implementation of Customer remote interface (part 2 of 6).
initialContext = new InitialContext();

Object object = initialContext.lookup("java:comp/env/ejb/Order");

OrderHome orderHome = (OrderHome)
PortableRemoteObject.narrow( object,
OrderHome.class );

// find Orders for this Customer
Collection orders =
orderHome.findByCustomerID( customerID );

Iterator iterator = orders.iterator();

// use list of Orders to build Order history
while ( iterator.hasNext() ) {
    Order order = (Order)
PortableRemoteObject.narrow(
iterator.next(), Order.class );

    // retrieve OrderModel for the Order
    OrderModel orderModel = order.getOrderModel();

    // add each OrderModel to Order history
    history.add( orderModel );
}

} // end try

// handle exception when finding Order records
catch (FinderException finderException) {
    throw new NoOrderHistoryException( "No order " +
"history found for the customer with userID " +
userID );
}

// handle exception when invoking Order EJB methods
catch (Exception exception) {
    exception.printStackTrace();
    throw new EJBException( exception );
}

return history;

} // end method getOrderHistory

// get password hint for Customer
public String getPasswordHint()
{
    return passwordHint;
}
private void setCustomerModel( CustomerModel customer )
{
    // set Customer data members to CustomerModel values
    userID = customer.getUserID();
    password = customer.getPassword();
    passwordHint = customer.getPasswordHint();
    firstName = customer.getFirstName();
    lastName = customer.getLastName();
    billingAddressID =
        customer.getBillingAddress().getAddressID();
    shippingAddressID =
        customer.getShippingAddress().getAddressID();
    creditCardName = customer.getCreditCardName();
    creditCardNumber = customer.getCreditCardNumber();
    creditCardExpirationDate =
        customer.getCreditCardExpirationDate();
}

// create Customer EJB using given CustomerModel
public Integer ejbCreate( CustomerModel customerModel )
    throws CreateException
{
    try {
        // retrieve unique value for primary key using
        // SequenceFactory EJB
        initialContext = new InitialContext();
        // look up SequenceFactory EJB
        Object object = initialContext.lookup(
            "java:comp/env/ejb/SequenceFactory" );
        SequenceFactoryHome sequenceFactoryHome =
            ( SequenceFactoryHome ) PortableRemoteObject.narrow(
                object, SequenceFactoryHome.class );
        // find sequence for Customer EJB
        SequenceFactory sequenceFactory =
            sequenceFactoryHome.findByPrimaryKey( "Customer" );
        // retrieve next available customerID
        customerID = sequenceFactory.getNextID();
        // create Address EJBs for billing and shipping
        // Addresses
        object = initialContext.lookup(
            "java:comp/env/ejb/Address" );
    }

Fig. 20.2 CustomerEJB implementation of Customer remote interface
(part 4 of 6).
AddressHome addressHome = (AddressHome)
    PortableRemoteObject.narrow(object, AddressHome.class);

    // get Customer's billing address
    AddressModel billingAddressModel =
        customerModel.getBillingAddress();

    // create Address EJB for billing Address
    Address billingAddress =
        addressHome.create(billingAddressModel);

    // set addressID in billing AddressModel
    billingAddressModel.setAddressID((Integer)
        billingAddress.getPrimaryKey());

    // get Customer's shipping address
    AddressModel shippingAddressModel =
        customerModel.getShippingAddress();

    // create Address EJB for shipping Address
    Address shippingAddress =
        addressHome.create(shippingAddressModel);

    // set addressID in shipping AddressModel
    shippingAddressModel.setAddressID((Integer)
        shippingAddress.getPrimaryKey());

    // use CustomerModel to set data for new Customer
    setCustomerModel(customerModel);

} // end try

    // handle exception when looking up, finding and using EJBS
    catch (Exception exception) {
        throw new CreateException(exception.getMessage());
    }

    // EJB container will return a remote reference
    return null;

} // end method ejbCreate

    // perform any necessary post-creation tasks
    public void ejbPostCreate(CustomerModel customer) {}
Method `getCustomerModel` (lines 39–95) constructs a `CustomerModel` whose data members contain values from the `CustomerEJB`. `AddressEJB` maintains address information for each `Customer`. The `CustomerEJB` stores the `addressID` for each `Address` (billing and shipping). Method `getCustomerModel` uses interface `AddressHome` to obtain the `Customer`’s `Address` EJBs (lines 65–78). For each `Address` EJB, method `getCustomerModel` obtains an `AddressModel` and adds it to the `CustomerModel`.

Method `getOrderHistory` (lines 98–149) builds a `Collection` that contains an `OrderModel` for each `Order` the `Customer` has placed. Method `findByCustomerID` of interface `OrderHome` returns a `Collection` of `Orders` for a given `Customer` (line 116). Lines 115–130 obtain the `Collection` of `Orders` and build an `Collection` of `OrderModels` to represent the `Customer`’s `Order` history.

Method `setCustomerModel` (lines 158–179) is a helper method for method `ejbCreate` (lines 182–251). Method `setCustomerModel` modifies the details of the `CustomerEJB`, using data in the `CustomerModel` argument. Lines 161–177 retrieve the data members of the `CustomerModel` and set the values of the `CustomerEJB` data members.

The EJB container invokes method `ejbCreate` (lines 182–251) when creating a new instance of the `Customer` EJB. Method `ejbCreate` takes a `CustomerModel` argu-
ment, with which it invokes method setCustomerModel to initialize the CustomerEJB data members (line 239). Method ejbCreate uses SequenceFactory method getNextID to generate a unique customerID for the new Customer (line 203). Each Customer has a billing address and a shipping address, each of which is stored in an Address EJB. Lines 215–236 create these Address EJBs, using data from the AddressModels given in the CustomerModel argument.

### 20.2.3 CustomerHome Interface

Interface CustomerHome (Fig. 20.3) provides create methods for creating new Customers and finder methods for finding existing Customers. Method create (lines 17–18) corresponds to method ejbCreate in the CustomerEJB implementation (Fig. 20.2). Method findByLogin (lines 21–22) returns a Customer remote reference for the Customer with the given userID and password. This method authenticates Customers when they attempt to log into the store. Method findByUserID (lines 25–26) returns the Customer with the given userID. Method findByPrimaryKey (lines 29–30) returns the Customer with the given customerID. The EJB container provides implementations for these methods, using SQL queries the developer provides when deploying the application.

```java
// CustomerHome.java
// CustomerHome is the home interface for entity EJB Customer.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core libraries
import java.rmi.RemoteException;

// Java standard extensions
import javax.ejb.*;

// Deitel Bookstore libraries
import com.deitel.advjhtp1.bookstore.model.*;

public interface CustomerHome extends EJBHome {
    // create Customer EJB using given CustomerModel
    public Customer create( CustomerModel customerModel )
        throws RemoteException, CreateException;

    // find Customer with given userID and password
    public Customer findByLogin( String userID, String pass )
        throws RemoteException, FinderException;

    // find Customer with given userID
    public Customer findByUserID( String userID )
        throws RemoteException, FinderException;

    // find Customer with given customerID
    public Customer findByPrimaryKey( String customerID )
        throws RemoteException, FinderException;

    // find Customer with given customerID
}
```

Fig. 20.3 CustomerHome interface for creating and finding Customer EJB instances (part 1 of 2).
CustomerModel (Fig. 20.4) is a model class for the Customer EJB. CustomerModel contains private data members (lines 17–29) for each public data member in the CustomerEJB implementation (Fig. 20.2). Class CustomerModel maintains references to AddressModel objects for the Customer's billing and shipping addresses (lines 24–25). CustomerModel has set and get methods for each of its properties, implements interface XMLGenerator and provides method getXML (lines 168–233), for generating an XML Element that describes the Customer.

```java
public class CustomerModel implements Serializable, XMLGenerator {
    // CustomerModel properties
    private Integer customerID;
    private String userID;
    private String password;
    private String passwordHint;
    private String firstName;
    private String lastName;
    private AddressModel billingAddress;
    private AddressModel shippingAddress;
    private String creditCardName;
    private String creditCardNumber;
    private String creditCardExpirationDate;

    // construct empty CustomerModel
    public CustomerModel() {}

    // set customer ID
```

Fig. 20.4 CustomerModel for serializing Customer data (part 1 of 5).
public void setCustomerID( Integer id )
{
    customerID = id;
}

// get customer ID
public Integer getCustomerID()
{
    return customerID;
}

// set user ID
public void setUserID( String id )
{
    userID = id;
}

// get user ID
public String getUserID()
{
    return userID;
}

// set password
public void setPassword( String customerPassword )
{
    password = customerPassword;
}

// get password
public String getPassword()
{
    return password;
}

// set password hint
public void setPasswordHint( String passwordHint )
{
    passwordHint = passwordHint;
}

// get password hint
public String getPasswordHint()
{
    return passwordHint;
}

// set first name
public void setFirstName( String name )
{
    firstName = name;
}
// get first name
public String getFirstName()
{
    return firstName;
}

// set last name
public void setLastName( String name )
{
    lastName = name;
}

// get last name
public String getLastName()
{
    return lastName;
}

// set billing address
public void setBillingAddress( AddressModel address )
{
    billingAddress = address;
}

// get billing address
public AddressModel getBillingAddress()
{
    return billingAddress;
}

// set shipping address
public void setShippingAddress( AddressModel address )
{
    shippingAddress = address;
}

// get shipping address
public AddressModel getShippingAddress()
{
    return shippingAddress;
}

// set name of credit card
public void setCreditCardName( String name )
{
    creditCardName = name;
}

// get name of credit card
public String getCreditCardName()
{
    return creditCardName;
}
public void setCreditCardNumber( String number )
{
    creditCardNumber = number;
}

public String getCreditCardNumber()
{
    return creditCardNumber;
}

public void setCreditCardExpirationDate( String date )
{
    creditCardExpirationDate = date;
}

public String getCreditCardExpirationDate()
{
    return creditCardExpirationDate;
}

public Element getXML( Document document )
{
    // create customer Element
    Element customer =
        document.createElement( "customer" );

    // create customerID Element
    Element temp = document.createElement( "customerID" );
    temp.appendChild( document.createTextNode( String.valueOf( getCustomerID() ) ) );
    customer.appendChild( temp );

    // create userID Element
    temp = document.createElement( "userID" );
    temp.appendChild( document.createTextNode( getUserID() ) );
    customer.appendChild( temp );

    // create firstName Element
    temp = document.createElement( "firstName" );
    temp.appendChild( document.createTextNode( getFirstName() ) );
    customer.appendChild( temp );

    // create lastName Element
    temp = document.createElement( "lastName" );
    customer.appendChild( temp );

Fig. 20.4 CustomerModel for serializing Customer data (part 4 of 5).
Chapter 20  Enterprise Java Case Study: Business Logic Part 2

20.3 Address Implementation

The application maintains a billing address and a shipping address for each Customer. Each address contains similar information (e.g., street address, city, state and zip code), so we abstract these two address types into a single Address EJB. Each Customer EJB stores an ID for the billing address and an ID for the shipping address.

temp.appendChild( document.createTextNode( getLastName() ) );
customer.appendChild( temp );

// create billingAddress Element
temp = document.createElement( "billingAddress" );
temp.appendChild( billingAddress.getXML( document ) );

// create shippingAddress Element
temp = document.createElement( "shippingAddress" );
temp.appendChild( shippingAddress.getXML( document ) );

// create creditCardName Element
temp = document.createElement( "creditCardName" );
temp.appendChild( document.createTextNode( getCreditCardName() ) );
customer.appendChild( temp );

// create creditCardNumber Element
temp = document.createElement( "creditCardNumber" );
temp.appendChild( document.createTextNode( getCreditCardNumber() ) );
customer.appendChild( temp );

// create creditCardExpirationDate Element
temp = document.createElement( "creditCardExpirationDate" );
temp.appendChild( document.createTextNode( getCreditCardExpirationDate() ) );
customer.appendChild( temp );

// create passwordHint Element
temp = document.createElement( "passwordHint" );
temp.appendChild( document.createTextNode( getPasswordHint() ) );
customer.appendChild( temp );

return customer;

} // end method getXML

Fig. 20.4  CustomerModel for serializing Customer data (part 5 of 5).

Figure 20.5, Fig. 20.6 and Fig. 20.7 list the deployment settings for entity EJB Customer. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods.
Customer General Deployment Settings

Bean Type: Entity
Enterprise Bean Class: com.deitel.advjhtp1.bookstore.ejb.CustomerEJB

Fig. 20.5 Customer general deployment settings.

Customer Entity and Deployment Settings

Persistence Management: Container-Managed Persistence
Primary Key Class: java.lang.Integer
Primary Key Field Name: customerID
Database JNDI Name: jdbc/Bookstore

Method findByUserID SQL Statement:
SELECT customerID FROM Customer WHERE userID = ?1

Method findByLogin SQL Statement:
SELECT customerID FROM Customer WHERE userID = ?1 AND password = ?2

Method ejbStore SQL Statement:
UPDATE Customer SET billingAddressID = ?,
creditCardExpirationDate = ?, creditCardName = ?,
creditCardNumber = ?, firstName = ?, lastName = ?,
password = ?, passwordHint = ?,
shippingAddressID = ?, userID = ? WHERE
customerID = ?

Method ejbCreate SQL Statement:
INSERT INTO Customer ( billingAddressID,
creditCardExpirationDate, creditCardName,
creditCardNumber, customerID, firstName, lastName,
password, passwordHint, shippingAddressID,

Fig. 20.6 Customer entity and deployment settings (part 1 of 2).
### Customer Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Method</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ejb-Remove</code></td>
<td><code>DELETE FROM Customer WHERE customerID = ?</code></td>
</tr>
<tr>
<td><code>findByPrimaryKey</code></td>
<td><code>SELECT customerID FROM Customer WHERE customerID = ?</code></td>
</tr>
<tr>
<td><code>ejb-Load</code></td>
<td><code>SELECT billingAddressID, creditCardExpirationDate, creditCardName, creditCardNumber, firstName, lastName, password, passwordHint, shippingAddressID, userID FROM Customer WHERE customerID = ?</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Create</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>CREATE TABLE Customer ( billingAddressID INTEGER, creditCardExpirationDate VARCHAR( 255 ), creditCardName VARCHAR( 255 ), creditCardNumber VARCHAR( 255 ), customerID INTEGER, firstName VARCHAR( 255 ), lastName VARCHAR( 255 ), password VARCHAR( 255 ), passwordHint VARCHAR( 255 ), shippingAddressID INTEGER, userID VARCHAR( 255 ), CONSTRAINT pk_Customer PRIMARY KEY ( customerID ))</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table Delete</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>DROP TABLE Customer</code></td>
</tr>
</tbody>
</table>

**Fig. 20.6** Customer entity and deployment settings (part 2 of 2).

### Customer EJB References

<table>
<thead>
<tr>
<th>Coded Name</th>
<th>Type</th>
<th>Home</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ejb/Order</code></td>
<td>Entity</td>
<td>com.deitel.advjhtp1.bookstore.ejb.OrderHome</td>
</tr>
<tr>
<td><code>ejb/SequenceFactory</code></td>
<td>Entity</td>
<td>com.deitel.advjhtp1.bookstore.ejb.SequenceFactory-Home</td>
</tr>
<tr>
<td><code>ejb/Address</code></td>
<td></td>
<td>com.deitel.advjhtp1.bookstore.ejb.SequenceFactory</td>
</tr>
</tbody>
</table>

**Fig. 20.7** Customer EJB References (part 1 of 2).
20.3.1 Address Remote Interface

The Address remote interface (Fig. 20.8) has set and get methods for updating data in and retrieving data from the Address EJB. Method getAddressModel (lines 17–18) constructs an AddressModel that contains the details of a particular Address EJB.

20.3.2 AddressEJB Implementation

AddressEJB (Fig. 20.9) is the implementation of the Address remote interface. Class AddressEJB contains public, container-managed data members for the first and last names of the contact person at the Address, as well as the street address, city, state, zip code, country and phone number (lines 24–33).
```java
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.util.*;
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

// Deitel packages
import com.deitel.advjhtp1.bookstore.model.*;
import com.deitel.advjhtp1.bookstore.exceptions.*;

public class AddressEJB implements EntityBean {
    private EntityContext entityContext;

    // container-managed fields
    public Integer addressID;
    public String firstName;
    public String lastName;
    public String streetAddressLine1;
    public String streetAddressLine2;
    public String city;
    public String state;
    public String zipCode;
    public String country;
    public String phoneNumber;

    // get AddressModel
    public AddressModel getAddressModel() {
        AddressModel address = new AddressModel();

        // populate AddressModel fields with Address EJB data members
        address.setAddressID( addressID );
        address.setFirstName( firstName );
        address.setLastName( lastName );
        address.setStreetAddressLine1( streetAddressLine1 );
        address.setStreetAddressLine2( streetAddressLine2 );
        address.setCity( city );
        address.setState( state );
        address.setZipCode( zipCode );
        address.setCountry( country );
        address.setPhoneNumber( phoneNumber );

        return address;
    }
}
```

Fig. 20.9 AddressEJB implementation of Address remote interface (part 2 of 4).
private void setAddressModel( AddressModel address )
{
    // update Address' data members using values provided in the AddressModel
    firstName = address.getFirstName();
    lastName = address.getLastName();
    streetAddressLine1 = address.getStreetAddressLine1();
    streetAddressLine2 = address.getStreetAddressLine2();
    city = address.getCity();
    state = address.getState();
    zipCode = address.getZipCode();
    country = address.getCountry();
    phoneNumber = address.getPhoneNumber();
}

// create Address EJB using given AddressModel
public Integer ejbCreate( AddressModel address )
throws CreateException
{
    // retrieve unique value for primary key using SequenceFactory EJB
    try {
        Context initialContext = new InitialContext();

        // look up SequenceFactory EJB
        Object object = initialContext.lookup( "java:comp/env/ejb/SequenceFactory" );

        SequenceFactoryHome sequenceFactoryHome =
            ( SequenceFactoryHome ) PortableRemoteObject.narrow( object, SequenceFactoryHome.class );

        // find sequence for Address EJB
        SequenceFactory sequenceFactory =
            sequenceFactoryHome.findByPrimaryKey( "Address" );

        // retrieve next available addressID
        addressID = sequenceFactory.getNextID();

        // set addressID for Address (primary key)
        address.setAddressID( addressID );

        // use AddressModel to set data for new Address
        setAddressModel( address );
    }
    catch ( Exception exception ) {
        throw new CreateException( exception.getMessage() );
    }
}

Fig. 20.9 AddressEJB implementation of Address remote interface (part 3 of 4).
Method `getAddressModel` (lines 35–55) constructs an `AddressModel`, sets its properties to the values of the `Address` EJB’s public data members and returns the `AddressModel` to the caller. Method `setAddressModel` (lines 58–72) is a utility method that takes an `AddressModel` argument and updates the values in the `AddressEJB`’s data members.

The EJB container invokes method `ejbCreate` (lines 75–115) to create a new `AddressEJB`. Each `Address` must have a unique `addressID` for its primary key.
SequenceFactory method `getNextID` (line 97) generates this unique `addressID`. Line 100 sets the `addressID` value in the `AddressModel`. Line 103 passes the `AddressModel` to method `setAddressModel` to complete initialization of the `AddressEJB`.

### 20.3.3 AddressHome Interface

Interface `AddressHome` (Fig. 20.10) provides methods for creating and finding `Address` EJBs. Method `create` (lines 18–19) takes an `AddressModel` argument. The EJB container invokes method `ejbCreate` (Fig. 20.9) when a client invokes method `create`. Method `findByPrimaryKey` (lines 22–23) locates an existing `Address` EJB, using its `addressID` primary key, and returns a remote reference to the `Address`.

### 20.3.4 AddressModel

Class `AddressModel` (Fig. 20.11) is a model class that implements interface `XMLGenerator` and method `getXML` to generate an XML description of an `Address`. Class `AddressModel` contains properties (lines 17–149) for each public data member in the `AddressEJB` implementation (Fig. 20.9). Method `getXML` (lines 152–212) builds an XML `Element` that contains child `Elements` for each of the `AddressModel`'s properties.

```java
// AddressHome.java
// AddressHome is the home interface for entity EJB Address.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core libraries
import java.rmi.RemoteException;

// Java standard extensions
import javax.ejb.EJBHome;
import javax.ejb.*;

// Deitel Bookstore libraries
import com.deitel.advjhtp1.bookstore.model.*;

public interface AddressHome extends EJBHome {
    // create Address EJB using given AddressModel
    public Address create( AddressModel address )
        throws RemoteException, CreateException;

    // find Address with given addressID
    public Address findByPrimaryKey( Integer addressID )
        throws RemoteException, FinderException;
}
```

Fig. 20.10 `AddressHome` interface for creating and finding `Address` EJB instances.
// AddressModel.java
// AddressModel represents a Customer's address, including
// street, city, state and zip code.
package com.deitel.advjhtp1.bookstore.model;

// Java core libraries
import java.io.*;
import java.util.*;

// third-party libraries
import org.w3c.dom.*;

public class AddressModel implements Serializable, XMLGenerator {

    // AddressModel properties
    private Integer addressID;
    private String firstName;
    private String lastName;
    private String streetAddressLine1;
    private String streetAddressLine2;
    private String city;
    private String state;
    private String zipCode;
    private String country;
    private String phoneNumber;

    // construct empty AddressModel
    public AddressModel() {} 

    // set addressID
    public void setAddressID( Integer id )
    {
        addressID = id;
    }

    // get addressID
    public Integer getAddressID()
    {
        return addressID;
    }

    // set first name
    public void setFirstName( String name )
    {
        firstName = name;
    }

    // get first name
    public String getFirstName()
    {
        return firstName;
    }

    // set street address line one
    public void setStreetAddressLine1( String address )
    {
        streetAddressLine1 = address;
    }

    // get street address line one
    public String getStreetAddressLine1()
    {
        return streetAddressLine1;
    }

    // set street address line two
    public void setStreetAddressLine2( String address )
    {
        streetAddressLine2 = address;
    }

    // get street address line two
    public String getStreetAddressLine2()
    {
        return streetAddressLine2;
    }

    // set city
    public void setCity( String city )
    {
        this.city = city;
    }

    // get city
    public String getCity()
    {
        return city;
    }

    // set state
    public void setState( String state )
    {
        this.state = state;
    }

    // get state
    public String getState()
    {
        return state;
    }

    // set zip code
    public void setZipCode( String zipCode )
    {
        this.zipCode = zipCode;
    }

    // get zip code
    public String getZipCode()
    {
        return zipCode;
    }

    // set country
    public void setCountry( String country )
    {
        this.country = country;
    }

    // get country
    public String getCountry()
    {
        return country;
    }

    // set phone number
    public void setPhoneNumber( String phoneNumber )
    {
        this.phoneNumber = phoneNumber;
    }

    // get phone number
    public String getPhoneNumber()
    {
        return phoneNumber;
    }

    // serialize AddressModel to XML
    public String toXML()
    {
        return null;
    }

    // deserialize XML into AddressModel
    public void fromXML( String xml )
    {
        return;
    }

    // other methods...
}

Fig. 20.11 AddressModel for serializing Address EJB data (part 1 of 4).
// set last name
public void setLastName( String name )
{
    lastName = name;
}

// get last name
public String getLastName()
{
    return lastName;
}

// set first line of street address
public void setStreetAddressLine1( String address )
{
    streetAddressLine1 = address;
}

// get first line of street address
public String getStreetAddressLine1()
{
    return streetAddressLine1;
}

// set second line of street address
public void setStreetAddressLine2( String address )
{
    streetAddressLine2 = address;
}

// get second line of street address
public String getStreetAddressLine2()
{
    return streetAddressLine2;
}

// set city
public void setCity( String addressCity )
{
    city = addressCity;
}

// get city
public String getCity()
{
    return city;
}

// set state
public void setState( String addressState )
{
    state = addressState;
}

Fig. 20.11 AddressModel for serializing Address EJB data (part 2 of 4).
// get state
public String getState()
{
    return state;
}

// set zip code
public void setZipCode( String zip )
{
    zipCode = zip;
}

// get zip code
public String getZipCode()
{
    return zipCode;
}

// set country
public void setCountry( String addressCountry )
{
    country = addressCountry;
}

// get country
public String getCountry()
{
    return country;
}

// set phone number
public void setPhoneNumber( String phone )
{
    phoneNumber = phone;
}

// get phone number
public String getPhoneNumber()
{
    return phoneNumber;
}

// build XML representation of Customer
public Element getXML( Document document )
{
    // create address Element
    Element address = document.createElement( "address" );
    
    // create firstName Element
    Element temp = document.createElement( "firstName" );
    temp.appendChild( document.createTextNode( getFirstName() ) );
    address.appendChild( temp );

    // create lastName Element

Fig. 20.11 AddressModel for serializing Address EJB data (part 3 of 4).
Figure 20.12, Fig. 20.13 and Fig. 20.14 show the deployment settings for entity EJB Address. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods.
Address General Deployment Settings

<table>
<thead>
<tr>
<th>Bean Type</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise</td>
<td>com.deitel.advjhtpl.bookstore.ejb.AddressEJB</td>
</tr>
<tr>
<td>Bean Class</td>
<td>com.deitel.advjhtpl.bookstore.ejb.AddressHome</td>
</tr>
<tr>
<td>Home Interface</td>
<td>com.deitel.advjhtpl.bookstore.ejb.Address</td>
</tr>
<tr>
<td>Remote Interface</td>
<td>com.deitel.advjhtpl.bookstore.ejb.Address</td>
</tr>
</tbody>
</table>

Fig. 20.12 **Address General** deployment settings.

Address Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Persistence Management</th>
<th>Container-Managed Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key Class</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>Primary Key Field Name</td>
<td>addressID</td>
</tr>
<tr>
<td>Database JNDI Name</td>
<td>jdbc/Bookstore</td>
</tr>
<tr>
<td>Method ejb-Store SQL Statement</td>
<td>UPDATE Address SET city = ?, country = ?, firstName = ?, lastName = ?, phoneNumber = ?, state = ?, streetAddressLine1 = ?, streetAddressLine2 = ?, zipCode = ? WHERE addressID = ?</td>
</tr>
<tr>
<td>Method ejb-Create SQL Statement</td>
<td>INSERT INTO Address ( addressID, city, country, firstName, lastName, phoneNumber, state, streetAddressLine1, streetAddressLine2, zipCode ) VALUES ( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )</td>
</tr>
<tr>
<td>Method ejb-Remove SQL Statement</td>
<td>DELETE FROM Address WHERE addressID = ?</td>
</tr>
<tr>
<td>Method findByPrimaryKey SQL Statement</td>
<td>SELECT addressID FROM Address WHERE addressID = ?</td>
</tr>
<tr>
<td>Method ejb-Load SQL Statement</td>
<td>SELECT city, country, firstName, lastName, phoneNumber, state, streetAddressLine1, streetAddressLine2, zipCode FROM Address WHERE addressID = ?</td>
</tr>
</tbody>
</table>

Fig. 20.13 **Address** entity and deployment settings (part 1 of 2).
20.4 SequenceFactory Implementation

One of the fundamental concepts in relational databases is that of a primary key, which uniquely identifies a row in a database table. The primary key is needed to define relationships between tables in the database. For example, in our case study, each Order relates to a Customer by storing the Customer table’s primary key—customerID—as a field (called a foreign key) in the Order table. The customerID is guaranteed to be unique, so it can be used to determine which Customer placed an Order. The Customer, Order and Address EJBs all have SequenceFactory records from which these EJBs can obtain primary keys.

20.4.1 SequenceFactory Remote Interface

Interface SequenceFactory (Fig. 20.15) is the remote interface for the SequenceFactory EJB. Method getNextID (line 15) returns the next available primary key.
20.4.2 SequenceFactoryEJB Implementation

Figure 20.16 shows the SequenceFactoryEJB implementation of the SequenceFactory remote interface. The EJB container manages the synchronization of the public data members (lines 19–22) with the database.

Method getNextOrderID (lines 25–31) calculates the next available unique orderID by incrementing the current value of the orderID (line 27). This new value is saved in the EJB (line 28) and returned to the caller (line 30). Method getNextCustomerID, (lines 34–40) increments the current value of the customerID field and returns the value to the caller (line 39). Method getNextAddressID (lines 43–49) increments the value of the addressID field and returns the value to the caller on line 48.

For the SequenceFactory EJB to calculate unique orderIDs, customerIDs and addressIDs properly, there must be only one SequenceFactory EJB instance. If there were more than one SequenceFactory EJB instance, duplicate orderIDs, customerIDs or addressIDs could be generated. SequenceFactory EJB clients should use only method findSequenceFactory of interface SequenceFactoryHome to obtain the correct SequenceFactory EJB instance. The deployer must specify an SQL query that will return the same SequenceFactory record each time a method findSequenceFactory is invoked.
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class SequenceFactoryEJB implements EntityBean {
  private EntityContext entityContext;

  // container-managed fields
  public String tableName;   // table name for ID sequence
  public Integer nextID;     // next available unique ID

  // get next available orderID
  public Integer getNextID()
  {
    // store nextID for returning to caller
    Integer ID = new Integer( nextID.intValue() );

    // increment ID to produce next available unique ID
    nextID = new Integer( ID.intValue() + 1 );

    return ID;
  }

  // set entity context
  public void setEntityContext( EntityContext context )
  {
    entityContext = context;
  }

  // unset entity context
  public void unsetEntityContext()
  {
    entityContext = null;
  }

  // activate SequenceFactory EJB instance
  public void ejbActivate()
  {
    tableName = ( String ) entityContext.getPrimaryKey();
  }

  // passivate SequenceFactory EJB instance
  public void ejbPassivate()
  {
    tableName = null;
  }

  // remove SequenceFactory EJB instance
  public void ejbRemove() {}

  // store SequenceFactory EJB data in database
  public void ejbStore() {}

Fig. 20.16 SequenceFactoryEJB implementation of SequenceFactory remote interface (part 2 of 3).
20.4.3 SequenceFactoryHome Interface

Interface SequenceFactoryHome (Fig. 20.17) is the home interface for the SequenceFactory EJB. Method findByPrimaryKey (lines 15–16) returns a remote reference to the SequenceFactory EJB for the database table with the given name.

Figure 20.18 and Fig. 20.19 show the deployment settings for entity EJB SequenceFactory. In addition to the settings shown here, be sure to set the Transaction Type to Required for all business methods.

```java
// SequenceFactoryHome.java
// SequenceFactoryHome is the home interface for entity EJB
// SequenceFactory.
package com.deitel.advjhtp1.bookstore.ejb;

// Java core packages
import java.rmi.RemoteException;

// Java extension packages
import javax.ejb.*;

public interface SequenceFactoryHome extends EJBHome {
    // find SequenceFactory with given primary key
    public SequenceFactory findByPrimaryKey( String tableName )
        throws RemoteException, FinderException;
}
```

Fig. 20.17 SequenceFactoryHome interface for finding SequenceFactory EJB instances.

<table>
<thead>
<tr>
<th>Bean Type</th>
<th>Entity</th>
<th>Enterprise Bean Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td></td>
<td>com.deitel.advjhtp1.bookstore.ejb.SequenceFactoryEJB</td>
</tr>
</tbody>
</table>

Fig. 20.18 SequenceFactory general deployment settings (part 1 of 2).
### SequenceFactory General Deployment Settings

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home Interface</td>
<td><code>com.deitel.advjhtp1.bookstore.ejb.SequenceFactoryHome</code></td>
</tr>
<tr>
<td>Remote Interface</td>
<td><code>com.deitel.advjhtp1.bookstore.ejb.SequenceFactory</code></td>
</tr>
</tbody>
</table>

**Fig. 20.18** SequenceFactory general deployment settings (part 2 of 2).

### SequenceFactory Entity and Deployment Settings

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence Management</td>
<td>Container-Managed Persistence</td>
</tr>
<tr>
<td>Primary Key Class</td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>Primary Key Field Name</td>
<td><code>tableName</code></td>
</tr>
<tr>
<td>Database JNDI Name</td>
<td><code>jdbc/Bookstore</code></td>
</tr>
<tr>
<td>Method <code>ejb-Store</code> SQL Statement</td>
<td><code>UPDATE SequenceFactory SET nextID = ?</code> WHERE tableName = ?</td>
</tr>
<tr>
<td>Method <code>ejb-Create</code> SQL Statement</td>
<td><code>INSERT INTO SequenceFactory ( nextID, tableName )</code> VALUES ( ?, ? )</td>
</tr>
<tr>
<td>Method <code>ejb-Remove</code> SQL Statement</td>
<td><code>DELETE FROM SequenceFactory WHERE tableName = ?</code></td>
</tr>
<tr>
<td>Method <code>find-ByPrimaryKey</code> SQL Statement</td>
<td><code>SELECT tableName FROM SequenceFactory</code> WHERE tableName = ?</td>
</tr>
<tr>
<td>Method <code>ejb-Load</code> SQL Statement</td>
<td><code>SELECT nextID FROM SequenceFactory</code> WHERE tablename = ?</td>
</tr>
<tr>
<td>Table Create SQL Statement</td>
<td><code>CREATE TABLE SequenceFactory ( nextID INTEGER, tableName VARCHAR( 255 ), CONSTRAINT pk_SequenceFactory PRIMARY KEY ( tableName ) )</code></td>
</tr>
<tr>
<td>Table Delete SQL Statement</td>
<td><code>DROP TABLE SequenceFactory</code></td>
</tr>
</tbody>
</table>

**Fig. 20.19** SequenceFactory entity and deployment settings.
20.5 Deitel Bookstore Application Deployment with J2EE

Deploying the components of the Deitel Bookstore on the Java 2 Enterprise Edition (J2EE) Reference Implementation requires the use of the Application Deployment Tool. The following steps walk through the process of deploying the Order EJB. The deployment process for the other entity EJBs is similar. For general instructions on deploying stateful session EJBs, such as the ShoppingCart EJB, please refer to Chapter 14, Session EJBs and Distributed Transactions. For general instructions on deploying Java servlets, please refer to Chapter 11, Case Study: Servlet and JSP Bookstore.

20.5.1 Deploying Deitel Bookstore CMP Entity EJBs

To begin deploying the entity EJBs for the Deitel Bookstore, select the New Enterprise Bean... menu item in the Application Deployment Tool’s File menu (Fig. 20.20) to begin deploying the EJB. You then will be presented with a wizard-style interface for creating the EJB JAR file (Fig. 20.21). The JAR Display Name field contains the text that will appear for this EJB JAR in the Application Deployment Tool, but has no effect on the deployment of the application. Click the Add... button next to the Contents field to add the class files for the EJB to the JAR.

To add the EJB class files to the EJB JAR file, you must specify the Root Directory that contains the class’ package structure (Fig. 20.22). For example, the Order EJB is in the package com.deitel.advjhtp1.bookstore.ejb. If the compiled class file is placed in the directory D:\BookStore\com\deitel\advjhtp1\bookstore\ejb, select D:\BookStore as the Root Directory. Click the Browse... button to use a file selection dialog to select the Root Directory.
Once you have selected the proper **Root Directory**, select the class files for the EJB remote interface, home interface, implementation and other classes that the EJB requires (e.g., `OrderModel.class`, `XMLGenerator.class`, application-specific exception classes, etc.). By holding down the **CTRL** key, you may select multiple files at once. Click the **Add** button to add the selected class files to the EJB JAR and click **OK** (Fig. 20.23). Figure 20.24 shows the results of adding the class files for the **Order** EJB to the EJB JAR file.
Once you have added the class files to the EJB JAR file, you must specify the class files that contain the remote interface, the home interface and the EJB implementation (Fig. 20.25). Select the appropriate class from the drop-down list as shown in Fig. 20.25. The **Order** EJB is an entity bean, so select the **Entity** radio button under the **Bean Type** heading (Fig. 20.26).
The **Order** EJB uses container-managed persistence to synchronize its data with the corresponding database table. In the following step of the EJB wizard (Fig. 20.27), select the **Container-Managed Persistence** radio button and place checkmarks next to each of the container-managed fields. Specify the complete class name of the primary-key class (including its package name) in the **Primary Key Class** field. For the **Order** EJB, enter
java.lang.Integer as the primary-key class. If your EJB uses a user-defined primary-key class (e.g., the OrderProduct EJB), you must also specify the complete package name (e.g., com.deitel.advjhtp1.bookstore.ejb.OrderProductPK). Select the field that contains the primary key from the Primary Key Field Name drop-down list (e.g., orderID).

If your EJB references other EJBs in its implementation, these EJBs must be specified in the Application Deployment Tool (Fig. 20.28). Click the Add button to add a new EJB reference. The Coded Name column corresponds to the String used to locate the EJB in the JNDI directory. For example, to locate the Product EJB, we use the String java:comp/env/ejb/Product. The corresponding coded name is ejb/Product. Select the appropriate type for the EJB (i.e., Session or Entity) from the Type drop-down list. Provide the full class name (including the package name) for the home and remote interfaces in the Home and Remote columns. For example, in the Home column for the Product EJB, specify com.deitel.advjhtp1.bookstore.ejb.ProductHome. Enter the JNDI name for the referenced EJB in the JNDI Name field (e.g., Product).

J2EE application servers provide transaction management, the semantics of which can be specified when deploying an application. For each of the business methods, specify the appropriate Transaction Type (Fig. 20.29), as discussed in Chapter 15.

Figure 20.30 shows the XML descriptor that was generated by the previous steps. This XML descriptor can be used when deploying this application on any J2EE-compliant application server.

---

**Fig. 20.27** Configuring container-managed fields and primary-key class.

---
**Fig. 20.28** Specifying other EJBs referenced by this EJB.

**Fig. 20.29** Specifying **Container-Managed Transactions** for EJB business methods.
You must now configure the database to which the CMP entity bean will persist its data. Click the **Deployment Settings...** button in the **Entity** tab (Fig. 20.31).

Specify the JNDI name for the database in the **Database JNDI Name** field (Fig. 20.32). In the J2EE Reference Implementation, this value corresponds to the value specified in the `default.properties` configuration file (e.g., `jdbc/BookStore`). Once you have specified the JNDI name for the database, click the **Generate SQL Now** button to create the necessary SQL statements for the EJB finder and create methods.
You will be prompted to provide SQL \texttt{WHERE} clauses for any custom \textit{finder} methods specified in the EJB’s home interface (Fig. 20.33). For each method listed under \textbf{EJB Method} (e.g., \texttt{findByPrimaryKey}, \texttt{ejbStore}, \texttt{Table Create}, etc.) enter the appropriate SQL query from the tables in Chapters 19–20. For example, Fig. 20.6 lists the appropriate SQL queries for the \texttt{Customer} EJB.

\subsection*{20.5.2 Deploying Deitel Bookstore Servlets}

The servlets in the Deitel Bookstore take advantage of context and initialization parameters—which the deployer supplies when deploying the application—to facilitate the installation of new client types. Figure 20.35 lists each servlet, the value for that servlet’s \texttt{XSL_FILE} initialization parameter and that servlet’s alias.

In addition, you must set the Web Context for the servlet’s WAR file to the value \texttt{bookstore} (Fig. 20.36).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig2032.png}
\caption{Configuring EJB Database Settings.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig2033.png}
\caption{Dialog indicating methods that require \texttt{WHERE} clauses for SQL queries.}
\end{figure}
**Fig. 20.34** Specifying SQL query for method `findByCustomerID`.

**Fig. 20.35** Deployment settings for Deitel Bookstore servlets.

<table>
<thead>
<tr>
<th>Servlet</th>
<th>XSL_FILE Initialization</th>
<th>Parameter Value</th>
<th>Servlet Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddToCartServlet</td>
<td>error.xsl</td>
<td></td>
<td>AddToCart</td>
</tr>
<tr>
<td>RemoveFromCartServlet</td>
<td>error.xsl</td>
<td></td>
<td>RemoveFromCart</td>
</tr>
<tr>
<td>UpdateCartServlet</td>
<td>error.xsl</td>
<td></td>
<td>UpdateCart</td>
</tr>
<tr>
<td>ViewCartServlet</td>
<td>viewCart.xsl</td>
<td></td>
<td>ViewCart</td>
</tr>
<tr>
<td>CheckoutServlet</td>
<td>error.xsl</td>
<td></td>
<td>Checkout</td>
</tr>
<tr>
<td>ViewOrderServlet</td>
<td>viewOrder.xsl</td>
<td></td>
<td>ViewOrder</td>
</tr>
<tr>
<td>ViewOrderHistoryServlet</td>
<td>viewOrderHistory.xsl</td>
<td></td>
<td>ViewOrderHistory</td>
</tr>
<tr>
<td>GetAllProductsServlet</td>
<td>products.xsl</td>
<td></td>
<td>GetAllProducts</td>
</tr>
<tr>
<td>GetProductServlet</td>
<td>productDetails.xsl</td>
<td></td>
<td>GetProduct</td>
</tr>
<tr>
<td>ProductSearchServlet</td>
<td>products.xsl</td>
<td></td>
<td>ProductSearch</td>
</tr>
<tr>
<td>RegisterServlet</td>
<td>error.xsl</td>
<td></td>
<td>Register</td>
</tr>
<tr>
<td>LoginServlet</td>
<td>login.xsl</td>
<td></td>
<td>Login</td>
</tr>
<tr>
<td>GetPasswordHintServlet</td>
<td>passwordHint.xsl</td>
<td></td>
<td>GetPasswordHint</td>
</tr>
</tbody>
</table>
Recall that the servlets in the Deitel Bookstore read client configuration information from the \texttt{CLIENT\_LIST} servlet context parameter. Specify the value for this context parameter as shown in Fig. 20.37.

The servlets in the Deitel Bookstore utilize EJB business logic to maintain the customers shopping cart, create customer registrations, etc. To enable the servlets to access the EJBs, we must specify EJB references in the deployment tool. Figure 20.38 shows the necessary EJB references. Be sure to specify the JNDI name for each EJB (e.g., \texttt{Shopping-Cart}) and full class names for the home and remote interfaces.
The final step before deploying the Deitel Bookstore case study is to add the XSL transformation documents and other supporting files to the servlet WAR file. Figure 20.39 lists these supporting files, their relative paths in the servlet WAR file and a description of each.

<table>
<thead>
<tr>
<th>File Name(s)</th>
<th>Relative Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clients.xml</td>
<td>/</td>
<td>Configuration file for enabling client support.</td>
</tr>
<tr>
<td>index.html</td>
<td>/</td>
<td>Welcome page for XHTML clients.</td>
</tr>
<tr>
<td>login.html</td>
<td>/</td>
<td>Login form for XHTML clients.</td>
</tr>
<tr>
<td>registration.html</td>
<td>/</td>
<td>Registration form for XHTML clients.</td>
</tr>
<tr>
<td>index.wml</td>
<td>/</td>
<td>Welcome page for WML clients.</td>
</tr>
<tr>
<td>login.wml</td>
<td>/</td>
<td>Login form for WML clients.</td>
</tr>
<tr>
<td>default.css</td>
<td>/styles/</td>
<td>Cascading Style Sheet for XHTML clients.</td>
</tr>
<tr>
<td>*.jpg</td>
<td>/images/</td>
<td>Book cover images for bookstore products.</td>
</tr>
<tr>
<td>*.xsl</td>
<td>/XSLT/XHTML/</td>
<td>XSL transformations for XHTML clients.</td>
</tr>
<tr>
<td>navigation.xml</td>
<td>/XSLT/XHTML/</td>
<td>Navigation header for XHTML clients.</td>
</tr>
</tbody>
</table>
After completing the EJB and servlet configuration in the Application Deployment Tool, select Deploy Application from the Tools menu to deploy the application to the J2EE reference implementation application server. You can access the newly deployed application by opening the URL http://localhost:8000/bookstore/index.html in a Web browser or i-mode simulator, or the URL http://localhost:8000/bookstore/index.wml for a WML simulator.

This concludes our discussion of the Deitel Bookstore case study. This case study integrated several enterprise Java technologies into a substantial online application. In Chapter 21, Application Servers, we introduce three of the most popular J2EE-compliant commercial application servers—BEA’s WebLogic, IBM’s WebSphere and the iPlanet Application Server. We then discuss the steps necessary to deploy the Deitel Bookstore case study on BEA’s WebLogic and IBM’s WebSphere.

### Fig. 20.39 Supporting files for inclusion in servlet WAR file (part 2 of 2).

<table>
<thead>
<tr>
<th>File Name(s)</th>
<th>Relative Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.xsl</td>
<td>/XSLT/WML/</td>
<td>XSL transformations for WML clients.</td>
</tr>
<tr>
<td>*.xsl</td>
<td>/XSLT/cHTML/</td>
<td>XSL transformations for cHTML clients.</td>
</tr>
</tbody>
</table>
Objectives

- To introduce several popular commercial application servers.
- To introduce open-source alternatives to commercial application servers.
- To understand the requirements for J2EE-compliant application servers.
- To understand the differences among commercial application server implementations.
- To deploy the Deitel Bookstore Enterprise Java case study on two leading commercial application servers.

Can anything be so elegant as to have few wants, and to serve them one’s self?

Ralph Waldo Emerson

“Contrariwise,” continued Tweedledee, “if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.”

Lewis Carroll

_Eloquence is logic on fire._

Lyman Beecher

...a pool that nobody's fathomed the depth of, and paths threaded with flowers planted by the mind.

Katherine Mansfield
Chapter 21

21.1 Introduction

The Java 2 Enterprise Edition is a specification for enterprise runtime environments. Although Sun provides a reference implementation of this specification, real-world systems must use an application server from a commercial vendor. In this chapter, we introduce the three most popular J2EE-compliant, commercial application servers—BEA WebLogic, IBM WebSphere and iPlanet Application Server. We also introduce the JBoss open-source application server. We deploy the Deitel Bookstore application from Chapters 17–20 to demonstrate the portability of applications written for the J2EE specification. After reading this chapter you will understand the role of an Application Server in an enterprise application, and be able to deploy your own applications on commercial application servers.

21.2 J2EE Specification and Benefits

For many years, there did not exist a standard for application servers. Each application server vendor provided its own set of APIs and varying functionality. If a company wished to move its enterprise applications to a new application-server platform, that company’s developers would need to rewrite large amounts of code, resulting in a complex and expensive migration process. Sun Microsystems, along with a large community of application-server vendors, developed the Java 2 Enterprise Edition specification through the Java Community Process (Appendix F, Java Community Process). J2EE defines an application server platform and supporting APIs for building enterprise applications that are portable across application servers, and, because they use Java, across platforms. J2EE extends Java’s “Write Once, Run Anywhere™” principle to enterprise applications. J2EE facilitates portability among application servers by enabling deployers to specify server-dependant features, such as distributed transactions and database queries at deployment time. At the time of this writing, the J2EE specification was evolving through the Java Community Process. The current release is 1.2.1, and version 1.3 is in beta.
The J2EE specification can be broken down into several pieces, including API support, security, transaction management and deployment processes. An application server vendor is required to provide runtime support for the APIs of the J2EE platform. Figure 21.1 lists the specific API requirements for version 1.2 of the J2EE specification.3

21.3 Commercial Application Servers

To be J2EE certified, an application server must implement the minimum functionality that the J2EE specification defines. Application server vendors can provide functionality that goes beyond the J2EE specification to differentiate their products. For example, application servers can provide advanced deployment tools, enhanced security strength, higher performance, error recovery, etc. This section describes four popular application servers—BEA WebLogic, iPlanet Application Server, IBM WebSphere and JBoss.

21.3.1 BEA WebLogic 6.0

BEA Systems is currently the number one application-server provider in the world. WebLogic’s popularity is largely based upon its first-to-market advantage and its reputation with current enterprises. BEA provides a general-purpose application server, balancing speed with stability and solid support for various features beyond the J2EE specification.

<table>
<thead>
<tr>
<th>Required APIs</th>
<th>Web Containers</th>
<th>EJBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Data Base Connectivity(JDBC) 2.0 Extension</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Remote Method Invocation-Internet Inter-ORB Protocol(RMI-IIOP) 1.0</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Enterprise Java Beans(EJB) 1.1</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Servlets 2.2</td>
<td>required</td>
<td>N/A</td>
</tr>
<tr>
<td>Java Server Pages(JSP) 1.1</td>
<td>required</td>
<td>N/A</td>
</tr>
<tr>
<td>Java Messaging System(JMS) 1.0</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Java Naming and Directory Interface(JNDI) 1.2</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Java Transaction API (JTA) 1.0</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>JavaMail 1.1</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>Java Activation Framework(JAF) 1.0</td>
<td>required</td>
<td>required</td>
</tr>
</tbody>
</table>

Fig. 21.1 Application server required APIs.
WebLogic provides data pools, which eliminate the need to create new database connections for each client. Establishing database connections has considerable overhead—up to several seconds per connection for some database servers. By maintaining a pool of open connections and assigning these to clients as needed, WebLogic increases application performance. WebLogic also provides a mechanism for "hot" deployment. The application server continually checks a specified directory for new applications; if the application server finds a new application in that directory, WebLogic automatically deploys the application without restarting the server. WebLogic also will undeploy an application if the administrator removes that application from the deployment directory. Hot deployment increases server up time and makes deployment simpler. However, BEA does not recommend enabling hot deployment in production environments. Rather, hot deployment can be useful for developers when testing applications.

WebLogic uses clustering to increase availability through failover support for EJBs and Web components. Servers within a cluster enable redundancy—if a server conducting a transaction fails, another server can take over transparently without interrupting the transaction. This redundancy also facilitates load balancing. In a load-balanced environment, the application server distributes requests among several servers based on their load (i.e., how many requests each server already is handling). Load balancing helps prevent individual servers from failing under high numbers of requests.

For single-server environments, WebLogic provides multi-pooling—a service that distributes transactions among data sources. Whereas a connection pool is limited to a single data source; multi-pooling allows an application to access several pools, thus distributing requests among multiple data sources. Multi-pooling divides labor among data sources to provide a degree of load balancing within a single application server.

WebLogic is completely J2EE 1.2 compliant and already supports many requirements of the J2EE 1.3 beta specification. WebLogic includes a Web server, but no longer includes a graphical deployment tool, so deployers must code deployment descriptors manually. WebLogic also works with several popular Java Development environments, including JBuilder and Visual Café.

WebLogic also provides advanced security. User access and control are managed within WebLogic through Access control lists (ACLs)—ACLs provide an efficient method for managing users and permissions, WebLogic also includes SSL support and digital certificates.

21.3.2 iPlanet Application Server 6.0

iPlanet E-Commerce Solutions is an alliance between Netscape Communications and Sun Microsystems. iPlanet has created a fully J2EE-certified application server as a replacement for Netscape’s application server. iPlanet’s primary goals are speed, stability and full J2EE compliance. iPlanet integrates C++ with Java to produce a fast, scalable application server. iPlanet provides failover support, connection pooling and several unique features.

The Web connector controls load balancing in iPlanet application server. The Web connector manages communication between the application server and the Web server. The Web connector distributes requests among server instances based on server response time. For more control, the application server may handle its own load balancing. Requests are distributed according to a specific algorithm defined by the deployer in the configuration tool. iPlanet includes support for "sticky" load balancing—if a component is flagged as
“sticky,” then the normal load-balancing algorithm will be bypassed and the component 
will always be executed on the “sticky” machine. “Sticky” load balancing helps in situa-
tions where the overhead of creating connections between servers for a given EJB is greater 
than the time saved by load balancing.

iPlanet uses the Lightweight Directory Access Protocol (LDAP) to manage security. 
Users may be assigned group and individual permissions to access parts of applications. Configuring LDAP permissions adds additional complexity to the server configuration pro-
cess, but enables tight control over user permissions. iPlanet Application Server is inte-
grated with, and includes, the iPlanet Directory Server and the iPlanet Web Server.

At the time of this writing, iPlanet was about to release a new version of iPlanet Application Server. To download the latest version, please visit www.iplanet.com/ 
ias_deitel. When this new version of iPlanet is available, we will provide a complete 
description of the iPlanet deployment process for the Deitel Bookstore application on our 
Web site, www.deitel.com

21.3.3 IBM WebSphere Advanced Application Server 4.0

IBM WebSphere is a popular application server that is approaching BEA’s WebLogic in 
market share. Version 4.0 is a substantial improvement over previous versions, including 
simpler configuration options, faster response time and enhanced security. Version 4.0 pro-
vides a simple user interface for administration and deployment, and a focus on speed and 
scalability. WebSphere includes IBM’s version of the Apache Web server, failover sup-
port, data pooling, and user-level security controls.

21.3.4 JBoss 2.2.2 Application Server

JBoss, combined with the Apache Software Foundation’s Tomcat servlet container, cur-
rently is the only J2EE 1.2-compliant, open-source application server. JBoss is distributed 
distributed under the Lesser General Public License (LGPL). The source code for JBoss is freely avail-
able, and can be used to serve commercial applications. JBoss aims to maintain compliance 
with future J2EE specifications. Currently, JBoss includes most features of commercial ap-
plication servers. JBoss was one of the first application servers to support hot deployment, 
and it runs with a small memory footprint, which leaves more resources for applications. 2

JBoss lacks clustering support, which may not be a concern for many small to mid-
sized businesses, but prevents JBoss from competing in the commercial market. JBoss also 
does not provide graphical tools for deployment or configuration. Fortunately, the JBoss 
community consists of a large and generally helpful user base that will help with most con-
figuration issues quickly.

21.4 Deploying the Deitel Bookstore on BEA WebLogic

Configuring an application server can be a complex task. In this section, we guide you 
through the steps required to install and configure BEA WebLogic for the Deitel Bookstore 
case study. Please refer to e-docs.bea.com/wls/docs60/install/install-prg.html for installation instructions. For these configuration instructions we assume 
c: \bea as the home directory, Deitel as the WebLogic administration domain name, 
and bookstore as the server name. We also assume Cloudscape is installed in
c:\cloudscape_3.6 and our database is c:\cloudscape_3.6\databases\Bookstore. Depending on your security requirements you may wish to create a file named password.ini with the password you entered during installation. Placing password.ini in c:\bea\wlserver6.0\config\deitel\ will prevent WebLogic from asking for your password each time you start the application server.

First, we configure WebLogic to enable Cloudscape database support. Open the file c:\bea\wlserver6.0\config\CaseStudyDS\startWebLogic.cmd with a text editor, such as Notepad. Near the bottom of the file replace the line

```
set CLASSPATH=.;\lib\weblogic_sp.jar;\lib\weblogic.jar
```

with

```
set CLASSPATH=.;\lib\weblogic_sp.jar;\lib\weblogic.jar;
c:\cloudscape_3.6\lib\cloudscape.jar
```

which includes the Cloudscape packages in the CLASSPATH. The following line contains the parameters to start the server:

```
%JAVA_HOME%\bin\java -hotspot -ms64m -mx64m -classpath
%CLASSPATH% -Dweblogic.Domain=deitel
-Dweblogic.Name=bookstore -Dbea.home=C:\bea
-Djava.security.policy==C:\bea\wlserver6.0/lib/weblogic.policy
-Dweblogic.management.password=%WLS_PW% weblogic.Server
```

To simplify detection of our database, we specify property cloudscape.system.home with value c:/cloudscape_3.6/databases/ on the command line.

Next, run startWeblogic.cmd either by double clicking it, or typing its name at the console. If a console window loads, and, after a minute, presents text similar to:

```
<WebLogic Server started>
<Notice> <WebLogicServer> <SSLListenThread listening on port 7002>
<Notice> <WebLogicServer> <ListenThread listening on port 7001>
```

the application server is ready for configuration.

Use your Web browser to access localhost:7001/console. A window should prompt for your network password. Enter system under User Name and the password you specified during WebLogic installation. Your Web browser should display the page shown in Fig. 21.2. This is the administration console for managing most aspects of the application server.

We now configure the JDBC data pool and data source. In the main pane of the administration console, select Connection Pools under the JDBC heading. The top of the right pane should contain the link Create new JDBC Connection Pool. Configure the options as shown in Fig. 21.3. Specify BookstorePool for the pool Name, the URL jdbc:cloudscape:Bookstore and the Driver Classname COM.cloudscape.core.JDBCDriver. Finally, set the password and server properties to none. After entering these values, select Apply. You need not restart the server until we have made a few more changes.
Next, select the **Targets** tab—select **Bookstore** from available servers, click the arrow pointing right to add this connection pool to the bookstore server then click **Apply**. In the right pane select **CaseStudyDS > Services > JDBC > Data Sources** then,
Create new JDBC data source. Specify the name of the data source (e.g., `BookstoreDataSource`), enter `jdbc/Bookstore` as the JNDI name, `BookstorePool` as the pool name and select Create. Finally, assign `Bookstore` under the Targets tab and click Apply. Shut down the server, close the WebLogic command window and restart the server with `startWebLogic.cmd`.

We now create the deployment descriptors for our bookstore application. Extract the contents of `bookstore.ear` into a new directory. You may use any zip or jar utility to extract the files. To extract the files via the `jar` utility included with the JDK, enter the command

```
jar xvf bookstore.ear
```

Next, extract the contents of `ejb-jar-ic.jar` into a temporary directory named `ejb-jar`. Create a text file named `weblogic-ejb-jar.xml` and save it in the `ejb-jar\META-INF` directory. This file is the WebLogic-specific deployment descriptor. If you are not familiar with XML please refer to Appendix A, Creating Markup with XML.

The Deployment descriptor `webLogic-ejb-jar.xml` (Fig. 21.4) defines name caching, persistence, transaction options, and other options for the EJBs in the Deitel Bookstore. Lines 6–8 specify the DTD for the descriptor. We specify the WebLogic 5.1.0 doctype because our application uses EJB 1.1. Element `weblogic-ejb-jar` (line 11–356) contains deployment information for all EJBs in the EJB JAR. Element `weblogic-enterprise-bean` (line 13–71) contains deployment information for the `Customer` EJB. Element `ejb-name` (line 14) specifies the name of this bean. Element `ejb-name` must match the name found in `ejb-jar.xml` for WebLogic to correctly identify the current bean. Lines 17–21 contain the descriptions of caching properties. Element `max-beans-in-cache` (Line 18) defines the maximum number of active instances the container should allow. When this limit is reached, the EJB container will passivate idle EJB instances. Element `cache-strategy` (line 19) sets how EJBs should cache data. Valid values include `Read-Write` or `Read-Only`. `Read-Write`, the default value, allows clients to write information to the bean in a standard transaction, and the container invokes `ejbStore` at the completion of the transaction. `Read-Only` does not allow the `ejbStore` method to be called, but it does allow the bean to be updated by an underlying data source periodically, for example this may be useful as a stock ticker bean. Line 19 defines this bean as a standard transaction bean. Element `read-timeout-seconds` is not used for `Read-Write` beans, if this were a `Read-Only` bean, the value specifies the number of seconds between database update intervals. When set to 0, the Read-Only bean updates only when it is first created.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE weblogic-ejb-jar PUBLIC
"-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB//EN'
'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-jar.dtd'>
```

Fig. 21.4 `Weblogic-ejb-jar.xml` defines WebLogic deployment properties for Bookstore case study (part 1 of 8).
Fig. 21.4 **Weblogic-ejb-jar.xml** defines WebLogic deployment properties for Bookstore case study (part 2 of 8).
Weblogic-ejb-jar.xml defines WebLogic deployment properties for Bookstore case study (part 3 of 8).
Fig. 21.4  **Weblogic-ear-jar.xml** defines WebLogic deployment properties for Bookstore case study (part 4 of 8).
Weblogic-ejb-jar.xml defines WebLogic deployment properties for Bookstore case study (part 5 of 8).
<ejb-ref-name>ejb/SequenceFactory</ejb-ref-name>
<jndi-name>ejb/SequenceFactory</jndi-name>
</ejb-reference-description>
</reference-descriptor>

<!-- assigns JNDI name to this bean -->
<jndi-name>ejb/Address</jndi-name>
</weblogic-enterprise-bean> <!-- end Address descriptor -->

<!-- OrderProduct EJB weblogic deployment descriptor -->
<weblogic-enterprise-bean>
  <ejb-name>OrderProduct</ejb-name>
  <caching-descriptor>
    <max-beans-in-cache>100</max-beans-in-cache>
    <idle-timeout-seconds>20</idle-timeout-seconds>
    <cache-strategy>Read-Write</cache-strategy>
    <read-timeout-seconds>0</read-timeout-seconds>
  </caching-descriptor>
  <persistence-descriptor>
    <persistence-type>
      <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
    </persistence-type>
    <type-storage>
      META-INF/weblogic-cmp-rdbms-jar-orderProduct.xml
    </type-storage>
  </persistence-descriptor>

<!-- maps this bean to specific cmp descriptor -->
<persistence-descriptor>
  <persistence-use>
    <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
  </persistence-use>
</persistence-descriptor>

<!-- maps references to JNDI names of beans -->
<ejb-reference-description>
  <ejb-ref-name>ejb/Product</ejb-ref-name>
  <jndi-name>ejb/Product</jndi-name>
</ejb-reference-description>

<!-- assigns JNDI name of this bean -->
<jndi-name>ejb/OrderProduct</jndi-name>

Fig. 21.4  Weblogic-ejb-jar.xml defines WebLogic deployment properties for Bookstore case study (part 6 of 8).
<weblogic-enterprise-bean>

<!-- end OrderProduct Descriptor -->

<!-- Product EJB weblogic deployment descriptor -->
<weblogic-enterprise-bean>
  <ejb-name>Product</ejb-name>
  <!-- defines caching properties for EJB -->
  <caching-descriptor>
    <idle-timeout-seconds>20</idle-timeout-seconds>
    <cache-strategy>Read-Write</cache-strategy>
    <read-timeout-seconds>0</read-timeout-seconds>
  </caching-descriptor>
  <!-- maps this bean to its CMP descriptor -->
  <persistence-descriptor>
    <persistence-type>
      <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
    </persistence-type>
    <persistence-use>
      <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
    </persistence-use>
  </persistence-descriptor>
  <!-- defines transaction properties -->
  <transaction-descriptor>
    <trans-timeout-seconds>200</trans-timeout-seconds>
  </transaction-descriptor>
  <!-- assigns JNDI name -->
  <jndi-name>ejb/Product</jndi-name>
</weblogic-enterprise-bean> <!-- end Product descriptor -->

<!-- ShoppingCart EJB weblogic deployment descriptor -->
<weblogic-enterprise-bean>
  <ejb-name>ShoppingCart</ejb-name>
  <!-- defines caching properties, set to defaults -->
  <caching-descriptor>
    <max-beans-in-cache>100</max-beans-in-cache>
  </caching-descriptor>
</weblogic-enterprise-bean> <!-- end ShoppingCart descriptor -->

Fig. 21.4 Weblogic-ejb-jar.xml defines WebLogic deployment properties for Bookstore case study (part 7 of 8).
Element **persistence-descriptor** (Lines 24–42) defines the EJBs persistence properties. The **Customer** bean uses container-managed persistence (CMP). WebLogic requires each CMP bean to have its own separate XML descriptor, thus the persistence block in this code simply identifies the name and type of persistence-management descriptor. Element **persistence-type** (lines 26–34) contains the elements **type-identifier** (line 27–29), which must be **WebLogic_CMP_RDBMS**. Element **type-version** is 5.1.0 for EJB 1.1 and 6.0 for EJB 1.2. The version number must correspond to the **weblogic-ejb-jar** DTD. Element **type-storage** (lines 31–33) specifies the location of the CMP descriptor, a file we will create next. You may define multiple **persistence-types**, in the next block, **persistence-use** you choose which **persistence-type** to use. Lines 36–41 tell WebLogic to use the persistence given in the previous element. Currently, element **transaction-descriptor** contains only a
single option—\textbf{trans-timeout-seconds} (Line 46). If a transaction lasts longer than
the specified length, it will be rolled back. Element \textbf{reference-descriptor} (lines
50–66) maps each reference to the correct JNDI name. Each block contains \textbf{ejb-ref-
name} and \textbf{jndi-name} elements for resolving references to other EJBs.

The descriptors for each of the other entity EJBs follow the same format as the
descriptor for \textbf{Customer}. The only differences are the reference to the RDBMS persistence descriptor file and each bean’s EJB references.

Stateful session EJB \textbf{ShoppingCart} (lines 315–354) requires slightly different
deployment information. Stateful session beans do not require a RDBMS persistence
descriptor file. Instead they simply require a location in which to store persistent sessions
and a directory location in which to store passivated sessions. Element \textbf{stateful-ses-
sion-persistent-store-dir} within element \textbf{persistence-descriptor}
defines where the container should store passivated sessions.\footnote{Figure 21.5 lists some optional elements in \texttt{weblogic-ejb-jar.xml}. For a full
listing, please consult \texttt{edocs.bea.com}. If an element is not defined, WebLogic uses its
default value.}

The RDBMS descriptor specifies the interactions of an entity EJB with the database.
The descriptor specifies a connection pool and table name, maps EJB fields to database
fields, and defines custom finders for home interface methods.

<table>
<thead>
<tr>
<th>Parent Element</th>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caching-descriptor</td>
<td>max-beans-in-free-pool</td>
<td>Valid for stateless session EJBs, defines the maximum number of free beans to keep in the pool. The default is no limit.</td>
</tr>
<tr>
<td>caching-descriptor</td>
<td>initial-beans-in-free-pool</td>
<td>Valid for stateless session EJBs, defines the number of initial bean instances. The default is 0.</td>
</tr>
<tr>
<td>persistence-descriptor</td>
<td>is-modified-method-name</td>
<td>Method name to be called when EJB is stored, method must return a boolean value. If method returns \texttt{true} EJB is saved.</td>
</tr>
<tr>
<td>Parent Element</td>
<td>Element</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>persistence-descriptor</td>
<td>delay-updates-until-end-of-tx</td>
<td>When set to false, the beans database table is updated after every method. If true database is updated at the end of the transaction. The default is true.</td>
</tr>
<tr>
<td>persistence-descriptor</td>
<td>finders-call-ejbload</td>
<td>Valid for entity beans, the value true specifies that bean is loaded after it is first referenced with a finder method. Value false specifies that the bean is loaded when first invoked. The default is false.</td>
</tr>
<tr>
<td>persistence-descriptor</td>
<td>db-is-shared</td>
<td>Valid for entity beans, For value false bean assumes it has exclusive access to database and does not reload data. For value true data is reloaded before each transaction. The default is true.</td>
</tr>
<tr>
<td>reference-descriptor</td>
<td>resource-descriptor</td>
<td>Contains description of resource factories referenced in ejb-jar.xml</td>
</tr>
<tr>
<td>resource-descriptor</td>
<td>res-ref-name</td>
<td>Resource reference name found in ejb-jar.xml</td>
</tr>
<tr>
<td>resource-descriptor</td>
<td>jndi-name</td>
<td>Assigns a JNDI name for resource factory</td>
</tr>
</tbody>
</table>
Descriptor weblogic-cmp-rdbms-jar-address.xml (Fig. 21.6) follows the WebLogic version 5.1 DTD. Element weblogic-rdbms-bean (line 11) is the base element for the descriptor, there may be only one webLogic-rdbms-bean element per file. Element pool-name on line 14 must contain the name for a data pool already defined in WebLogic. The value of table-name must correspond to the table within the data source that this bean uses. In our case study, EJB Address writes to table Address (line 17). Element attribute-map contains the mappings between EJB fields and database fields defined in ejb-jar.xml. In our example the database field names are the same as the EJB field names, but this is not required. Defining mappings within a deployment descriptor allows deployers to customize EJBs to particular databases without changing any EJB code. Element object-link (line 21–24) contains the mappings between the EJB fields and database fields for zipCode. The bean field and database field names are specified within bean-field and dbms-column, respectively.

```
1  <?xml version = "1.0" encoding = "UTF-8" ?>
2  3  <!-- weblogic-cmp-rdbms-jar-address.xml rdbms deployment -->
4  5  <!-- descriptor for EJB Address. Defines database properties -->
```

Fig. 21.6  Weblogic-cmp-rdbms-jar-address.xml defines WebLogic CMP database properties for EJB Address (part 1 of 3).
<!DOCTYPE weblogic-rdbms-bean PUBLIC
'-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB RDBMS Persistence/
http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persistence.dtd'>

<!-- main block for rdbms descriptor -->
<weblogic-rdbms-bean>
  <!-- Sets data pool to BookstorePool -->
  <pool-name>BookstorePool</pool-name>
  <!-- Sets database table to ADDRESS -->
  <table-name>Address</table-name>
  <!-- maps EJB fields to database fields -->
  <attribute-map>
    <object-link>
      <bean-field>zipCode</bean-field>
      <dbms-column>zipCode</dbms-column>
    </object-link>
    <object-link>
      <bean-field>state</bean-field>
      <dbms-column>state</dbms-column>
    </object-link>
    <object-link>
      <bean-field>addressID</bean-field>
      <dbms-column>addressID</dbms-column>
    </object-link>
    <object-link>
      <bean-field>streetAddressLine2</bean-field>
      <dbms-column>streetAddressLine2</dbms-column>
    </object-link>
    <object-link>
      <bean-field>country</bean-field>
      <dbms-column>country</dbms-column>
    </object-link>
    <object-link>
      <bean-field>streetAddressLine1</bean-field>
      <dbms-column>streetAddressLine1</dbms-column>
    </object-link>
    <object-link>
      <bean-field>city</bean-field>
      <dbms-column>city</dbms-column>
    </object-link>
  </attribute-map>
</weblogic-rdbms-bean>

Fig. 21.6 Weblogic-cmp-rdbms-jar-address.xml defines WebLogic CMP database properties for EJB Address (part 2 of 3).
WebLogic XML descriptor `weblogic-cmp-rdbms-jar-Customer.xml` (Fig. 21.7) is the database deployment descriptor for the `Customer` EJB. The database descriptors closely mirror each other in terms of structure. For each EJB you must provide field mappings, the database table name and custom queries for finder methods.

**Fig. 21.6** Weblogic-cmp-rdbms-jar-address.xml defines WebLogic CMP database properties for EJB `Address` (part 3 of 3).

WebLogic XML descriptor `weblogic-cmp-rdbms-jar-Customer.xml` (Fig. 21.7) is the database deployment descriptor for the `Customer` EJB. The database descriptors closely mirror each other in terms of structure. For each EJB you must provide field mappings, the database table name and custom queries for finder methods.

**Fig. 21.7** WebLogic-cmp-rdbms-jar-Customer.xml defines WebLogic CMP database properties for EJB `CustomerEJB` (part 1 of 4).
<!-- field mapping for lastName -->
<object-link>
  <bean-field>lastName</bean-field>
  <dbms-column>lastName</dbms-column>
</object-link>

<!-- field mapping for userID -->
<object-link>
  <bean-field>userID</bean-field>
  <dbms-column>userID</dbms-column>
</object-link>

<!-- field mapping for creditCardNumber -->
<object-link>
  <bean-field>creditCardNumber</bean-field>
  <dbms-column>creditCardNumber</dbms-column>
</object-link>

</attribute-map>

<!-- list of custom finders -->
<finder-list>
  <!-- finder for findByUserID -->
  <finder>
    <method-name>findByUserID</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>

    <!-- get fields where userID matches argument string -->
    <finder-query>
      <![CDATA[( like userID $0 )]]>
    </finder-query>
  </finder>

  <!-- finder for findByLogin -->
  <finder>
    <method-name>findByLogin</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
      <method-param>java.lang.String</method-param>
    </method-params>

    <!-- fields where userID and password match arguments -->
    <finder-query>
      <![CDATA[(& ( like userID $0 ) ( like password $1 ))]]>
    </finder-query>
  </finder>
</finder-list>

Fig. 21.7 WebLogic-cmp-rdbms-jar-Customer.xml defines WebLogic CMP database properties for EJB CustomerEJB (part 3 of 4).
Element **finder-list** (lines 91–120) contains **finder** methods for the customer EJB. WebLogic requires a **finder** element for each customer **finder** method in the home interface. You must define the **finder** name, parameters, and the query to use. The value of element **method-name** (line 95) is a string that matches a **finder** method defined in the home interface. Element **method-params** (lines 96–98) contains element **method-param** (line 97), which specifies the fully qualified type of any passed parameters (e.g. `java.lang.String`). The **finder-query** is a query written in WebLogic Query Language (WQL). Be sure to place WQL expressions in **CDATA** sections to escape special characters.

Figure 21.8 lists operators and syntax examples in WQL. The operands of the expressions may be arguments, literals, **bean-fields** within this descriptor, or other expressions. The syntax $n$ signifies that the operator is an argument and $n$ corresponds to the order of the passed arguments, beginning at 0. Literals must always be enclosed in single quotes. Lines 107–118 define **findByLogin**, which takes two arguments and returns only the fields that match both.\(^5\)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
<td>( = ID $0 )</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>( &lt; price $0 )</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>( &gt; quantity $0 )</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>( &lt;= operand1 operand2 )</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>( &gt;= operand1 operand2 )</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>( ! (= quantity '0') )</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
<td>( &amp; (&lt; price $0 ) ( &gt; quantity $1 ) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>like</td>
<td>string equal to, % indicates wildcard</td>
<td>( like '%Java%' )</td>
</tr>
<tr>
<td>isNull</td>
<td>check for null reference</td>
<td>( isNull bookID )</td>
</tr>
<tr>
<td>isNotNull</td>
<td>check if not null</td>
<td>( isNotNull bookID )</td>
</tr>
</tbody>
</table>
Figure 21.9 through Fig. 21.12 contain the remaining CMP descriptors, which each follow the same basic structure.

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!-- weblogic-cmp-rdbms-jar-order.xml ejb descriptor for -->
<!-- OrderEJB defines rdbms properties for WebLogic -->
<!DOCTYPE weblogic-rdbms-bean PUBLIC
'-//BEA Systems, Inc./DTD WebLogic 5.1.0 EJB RDBMS Persistence/
/EN'
'http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persistence.dtd'>
<!-- element containing rdbms properties for OrderEJB -->
<weblogic-rdbms-bean>
  <!-- assigns this bean to pool named BookstorePool -->
  <pool-name>BookstorePool</pool-name>
  <!-- assigns this bean to table named CUSTOMERORDERS -->
  <table-name>CustomerOrders</table-name>
  <!-- element containing field mappings -->
  <attribute-map>
    <!-- field mapping for orderDate -->
    <object-link>
      <bean-field>orderDate</bean-field>
      <dbms-column>orderDate</dbms-column>
    </object-link>
    <!-- field mapping for shipped -->
    <object-link>
      <bean-field>shipped</bean-field>
      <dbms-column>shipped</dbms-column>
    </object-link>
  </attribute-map>
</weblogic-rdbms-bean>
```

Fig. 21.9  *Weblogic-cmp-rdbms-jar-order.xml* defines WebLogic CMP database properties for EJB *OrderEJB* (part 1 of 2).
<?xml version="1.0" encoding="UTF-8"?>
<!-- weblogic-cmp-rdbms-jar-orderProduct.xml ejb descriptor for -->
<!-- OrderProductEJB defines rdbms properties for WebLogic -->

<weblogic-rdbms-bean>

<!-- field mapping for customerId -->
<object-link>
  <bean-field>customerId</bean-field>
  <dbms-column>customerId</dbms-column>
</object-link>

<!-- field mapping for orderID -->
<object-link>
  <bean-field>orderID</bean-field>
  <dbms-column>orderID</dbms-column>
</object-link>

<attribute-map/>

<finder-list>

<!-- finder for findByCustomerID -->
<finder>
  <method-name>findByCustomerID</method-name>
  <method-params>
    <method-param>java.lang.Integer</method-param>
  </method-params>
</finder>

<!-- return enumeration where customerId = argument -->
<finder-query><![CDATA[( = customerId $0 )]]></finder-query>

</finder-list>

<!-- additional options -->
<options>
  <use-quoted-names>false</use-quoted-names>
</options>

</weblogic-rdbms-bean>

<!-- end rdbms descriptor for OrderEJB -->

Fig. 21.9 Weblogic-cmp-rdbms-jar-order.xml defines WebLogic CMP database properties for EJB OrderEJB (part 2 of 2).

Fig. 21.10 Weblogic-cmp-rdbms-jar-orderProduct.xml defines WebLogic CMP database properties for the OrderProduct EJB (part 1 of 3).
Weblogic-cmp-rdbms-jar-orderProduct.xml defines WebLogic CMP database properties for the **OrderProduct** EJB (part 2 of 3).
Chapter 21 Application Servers

Fig. 21.10  **weblogic-cmp-rdbms-jar-orderProduct.xml** defines WebLogic CMP database properties for the **OrderProduct** EJB (part 3 of 3).

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE weblogic-rdbms-bean PUBLIC
 '-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB RDBMS Persistence/
/EN'
 'http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persis-
tence.dtd'>
<!-- element containing rdbms properties for ProductEJB -->
<weblogic-rdbms-bean>
  <!-- assigns this bean to pool named BookstorePool -->
  <pool-name>BookstorePool</pool-name>
  <!-- assigns this bean to table named PRODUCT -->
  <table-name>Product</table-name>
  <!-- element containing field mappings -->
  <attribute-map>
    <!-- field mapping for pages -->
    <object-link>
      <bean-field>pages</bean-field>
      <dbms-column>pages</dbms-column>
    </object-link>
    <!-- field mapping for author -->
    <object-link>
      <bean-field>author</bean-field>
      <dbms-column>author</dbms-column>
    </object-link>
  </attribute-map>
</weblogic-rdbms-bean>

Fig. 21.11  **weblogic-cmp-rdbms-jar-product.xml** defines WebLogic CMP database properties for the **Product** EJB (part 1 of 3).
Fig. 21.11  *weblogic-cmp-rdbms-jar-product.xml* defines WebLogic CMP database properties for the **Product** EJB (part 2 of 3).
Fig. 21.11  *weblogic-cmp-rdbms-jar-product.xml* defines WebLogic
CMP database properties for the *Product* EJB (part 3 of 3).

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE weblogic-rdbms-bean PUBLIC
'"//BEA Systems, Inc./DTD WebLogic 5.1.0 EJB RDBMS Persistence/'
'http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persistence.dtd'>

<!-- element containing rdbms properties for SequenceFactoryEJB -->
<weblogic-rdbms-bean>

<!-- assigns this bean to pool named BookstorePool -->
<pool-name>BookstorePool</pool-name>

<!-- assigns this bean to table SEQUENCEFACTORY -->
<table-name>SequenceFactory</table-name>

<!-- element containing field mappings -->
<attribute-map>

<!-- field mapping for addressID -->
<object-link>
  <bean-field>tableName</bean-field>
  <dbms-column>tableName</dbms-column>
</object-link>

<attribute-map>

<!-- field mapping for addressID -->
<object-link>
  <bean-field>tableName</bean-field>
  <dbms-column>tableName</dbms-column>
</object-link>
```

Fig. 21.12  *Weblogic-cmp-rdbms-jar-sequence.xml* defines WebLogic
CMP database properties for the *SequenceFactory* EJB (part 1 of 2).
Finally, the EJB references in the servlets must be mapped to the JNDI names for each EJB. The deployment descriptor weblogic.xml in Web application directory WEB-INF defines these mappings. Figure 21.13 contains the mappings for the Bookstore servlets. Optional element description provides a simple description for the web application. Element reference-descriptor is the only required element for our application. This element maps references defined in web.xml to the JNDI names of the referenced EJBs. We have named the references the same as the JNDI names, but that is not required. Each reference description is contained within element ejb-reference-description. A mapping requires the ejb-ref-name and jndi-name contained within ejb-reference-description. Lines 22–29 map the ShoppingCart EJB reference to its JNDI name.
This completes the deployment descriptors for our case study application; you are almost ready to deploy. Our application uses Xalan version 2.1, so prepend \texttt{xalan.jar} and \texttt{xerces.jar} to the \texttt{CLASSPATH} in \texttt{startWeblogic.cmd}. Your final \texttt{classpath} entry should be:

\begin{verbatim}
set CLASSPATH=.;c:\xalan-j_2_1_0\xerces.jar;c:\xalan-j_2_1_0\xalan.jar;\lib\weblogic_sp.jar;\lib\weblogic.jar;c:\cloudscape_3.6\lib\cloudscape.jar
\end{verbatim}

Be sure to replace \texttt{c:\xalan-j_2_1_0} with the correct path to the Xalan JAR files. Execute \texttt{startWeblogic.cmd} and open your Web browser to \texttt{localhost:7001/bookstore} to access the bookstore.

\subsection*{21.5 Deploying the Deitel Bookstore on IBM WebSphere}

This section presents the configuration and deployment of the Deitel Bookstore application on IBM WebSphere 4.0. Consult the WebSphere documentation for specific information.
on installing the application server on your system. We assume that Cloudscape is installed in `C:\cloudscape_3.6`, the WebSphere application server is installed in `C:\WebSphere\AppServer` and the IBM HTTP Server (included with installation) is installed in `C:\IBM HTTP Server`. You also must copy `xalan.jar` and `xerces.jar` to `C:\WebSphere\AppServer\lib\ext`, which ensures that our application has access to the correct XML parser and XSL transformer.

Next, run the `StartServerBasic` script from the `WebSphere\AppServer\bin` directory to launch the application server. When the server completes loading, open the URL `localhost:9090/admin` in a Web browser. You may specify any user name in the login page—the name is used to track changes, not for security. To configure the JDBC driver and data source, select Resources > JDBC Drivers then click the New button from the right pane. Enter `c:\cloudscape_3.6\lib\cloudscape.jar` for the Server Class Path, `Bookstore` for the Name and `COM.cloudscape.core.LocalConnectionPoolDataSource` for the Implementation Class Name. After entering these values, click OK and save the configuration. Expand the recently created driver (Bookstore) and select Data Sources. Click the New button, and fill in the fields as follows. The Name is Bookstore, the JNDI Name is `jdbc/Bookstore`, the Database Name should be the full path to the database (e.g., `c:\cloudscape_3.6\databases\bookstore\`). Click OK and save the configuration.

Next, we must generate the deployment descriptors for WebSphere. WebSphere includes a graphical tool for creating deployment descriptors. Open the Application Assembly Tool by executing the assembly script in the `WebSphere/AppServer/bin` directory. When the tool loads, select the Existing tab from the welcome window. Then, enter the full path to `bookstore.ear`, or select browse to locate the file. Click Open to open the file and begin deployment configuration.

In the left pane, expand Bookstore > EJB modules > EJBs > Session Beans > Shopping Cart. Select EJB `ShoppingCart` and choose the Bindings tab from the main pane (the bottom right pane). Enter `ShoppingCart` as the JNDI Name. Select EJB References from the upper right pane. Be sure `ejb/Order` is selected and use the Link drop-down box to select the appropriate reference. You also must select the Bindings tab from the main pane and enter `Order` for the reference’s JNDI Name. Complete the same steps to map the reference `ejb/Product` to the Product EJB.

Select Bookstore > EJB Modules > EJBs > Entity Beans > Address. Select the Bindings tab from the main pane and enter the JNDI Name Address, and set the datasource JNDI Name `jdbc/Bookstore`. Select EJB References from the upper pane and map `ejb/SequenceFactory` to `SequenceFactory` with JNDI Name `SequenceFactory`. Configure the EJB references for the other EJBs using the same process.

After defining the JNDI names and EJB references for the Customer EJB select Method Extensions. You must define custom finder queries for methods `findByUserID` and `findByLogin`. To define a custom query, select each method, check the Finder Descriptor box and specify the appropriate WHERE clause (Fig. 21.14). The WHERE clause option will fill in the SELECT and FROM SQL statements for you—all you must define is the WHERE clause. WebSphere determines the table name from the EJB name, thus to specify a different table you must rename the EJB. When deploying the Deitel
Bookstore be sure to select the **Order** EJB and change the EJB name to **Customer-Orders**, which is the table for storing order information.

To configure the servlets in WebSphere you must select **Bookstore > Web Modules > Servlets > EJB references** and map each reference as you did for each EJB. After you map the references you can generate the code for deployment. Select the **File** menu and **Generate code for deployment**. Specify the **Deployed module location** text field to the path in which to save the deployed EAR. Enter the path to **xalan.jar** and **xerces.jar** in the **Dependent Classpath** field. The **Database type** is **Generic/SQL-92**, **Database Name** is **jdbc/Bookstore** and **Schema** is **APP**. Fill in these values then select the **Generate Now** button.

You may now deploy the application in the administration tool Web page (**localhost:9090/admin**). In the left pane, select **Nodes > Computer Name > Enterprise Applications**. Click the **Install** button then use the **Browse** button to locate the deployed EAR file (e.g., **Deployed_Bookstore.ear**) and click **Next**. The following page shows the JNDI name mapping—be sure to double check each value then select **Next**. The following page shows each EJB reference. The following page verifies the database mappings. Each field must contain the value **jdbc/Bookstore**. The **Database type** must be **Generic/SQL-92** and the **Schema** name is **APP**. The next page defines the servlet mapping to **default_host**. In the following page, uncheck the option to redeploy the application. Click **Next**, confirm the values and click **Finish** to deploy. After the application is deployed, click the link at the top of the page to regenerate the Web server plug-in configuration. Save the configuration and restart the server by running the **stopServer** script followed by the **startServerBasic** script. Finally, open the URL **localhost/bookstore** to browse the Deitel Bookstore.

### 21.6 Internet and World Wide Web Resources

**serverwatch.internet.com/appservers.html**
Application server news site, also includes reviews and comparisons of popular application servers.

**www.appserver-zone.com**
The Application Server Zone Web site provides technical articles, product comparisons and other information related to application servers.

**java.sun.com/j2ee**
Sun’s J2EE site, includes the J2EE specification, J2EE news and SDK download and support.
**SUMMARY**

- The Java 2 Enterprise Edition is a specification for enterprise runtime environments. Although Sun provides a reference implementation of this specification, real-world systems must use an application server from a commercial vendor.

- Sun Microsystems, along with a large community of application-server vendors, developed the Java 2 Enterprise Edition specification through the Java Community Process. J2EE defines an application server platform and supporting APIs for building enterprise applications that are portable across application servers, and, because they use Java, across platforms.

- The J2EE specification can be broken down into several pieces, including API support, security, transaction management and deployment processes. An application server vendor is required to provide runtime support for the APIs of the J2EE platform.

- To be J2EE certified, an application server must implement the minimum functionality that the J2EE specification defines. Application server vendors also can provide functionality that goes beyond the J2EE specification to differentiate their products.

- BEA provides a general-purpose application server, balancing speed with stability and solid support for various features beyond the J2EE specification, including data pools, “hot” deployment clustering, failover support for EJBs and Web components and load balancing.

- For single-server environments, WebLogic provides multi-pooling—a service that distributes transactions among data sources. Whereas a connection pool is limited to a single data source; multi-pooling allows an application to access several pools, thus distributing requests among multiple data sources.

- iPlanet E-Commerce Solutions is an alliance between Netscape Communications and Sun Microsystems. iPlanet’s primary goals are speed, stability and full J2EE compliance. iPlanet provides failover support, connection pooling and several unique features.

- The Web connector controls load balancing in iPlanet application server. The Web connector manages communication between the application server and the Web server. The Web connector distributes requests among server instances based on server response time.

- iPlanet includes support for “sticky” load balancing—if a component is flagged as “sticky,” then the normal load-balancing algorithm will be bypassed and the component will always be executed on the “sticky” machine.

- iPlanet uses the Lightweight Directory Access Protocol (LDAP) to manage security. Users may be assigned group and individual permissions to access parts of applications. Configuring LDAP permissions enables tight control over user permissions.

- IBM WebSphere is a popular application server that is approaching BEA’s WebLogic in market share. Version 4.0 provides a simple user interface for administration and deployment, and a focus on speed and scalability. WebSphere includes IBM’s version of the Apache Web server, failover support, data pooling, and user-level security controls.

- JBoss, combined with the Apache Software Foundation’s Tomcat servlet container, currently is the only J2EE 1.2-compliant, open-source application server. JBoss aims to maintain compliance with future J2EE specifications. Currently, JBoss includes most features of commercial application servers.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Access Control List (ACL)</th>
<th>BEA Weblogic</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-map</td>
<td>bean-field</td>
</tr>
<tr>
<td>cache-strategy</td>
<td>element</td>
</tr>
</tbody>
</table>
data pool
data source
dbms-column element
deployment descriptor
description element
ejb-name element
ejb-reference-description element
ejb-ref-name element
ejbStore method
failover
finder element
finder-list element
finder-query element
hot deployment
IBM WebSphere
iPlanet Application Server
J2EE specification
JBoss
jndi-name element
life cycle
load balancing
max-beans-in-cache element
method-name element
method-param element
method-params element
multi pool

WORKS CITED
Objectives

- To understand the Jini Technology Architecture.
- To be able to identify the major components of a Jini solution.
- To be able to implement Jini services and register those services with Jini lookup services.
- To be able to write a Jini client.
- To be able to use Jini helper classes to simplify service implementations.

*The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.*
Marcel Proust

*For, when with beauty we can virtue join,*
*We paint the semblance of a form divine.*
Matthew Prior

*...To tell your name the livelong day*  
*To an admiring bog.*
Emily Dickinson

*...And summer’s lease hath all too short a date.*
William Shakespeare
Chapter 22 Jini

22.1 Introduction

Many network devices provide services to network clients. For example, a network printer provides printing services to many clients by allowing them to share the printer. Similarly, a Web server provides a service by allowing many clients to access documents over a network. We can extend this idea of providing services beyond computer-based networks and into home-based networks. For example, when you arrive home, your car could use a wireless network to notify your home’s lighting service to turn on the lights over the walkway, so you can walk into your home safely. Each service mentioned here has a well-defined interface. The network-printing service provides an interface that enables applications to print documents. A Web server provides an HTTP interface that enables Web browsers to download documents. Your home’s lighting service provides an interface that enables other devices on the network to turn the lights on and off.

To use a service, a client must be able to discover that a service exists and must know the interface for interacting with the service. For example, your car must be able to discover that your home provides a lighting service and must know the service’s interface to interact with the lighting service. However, the car need not know the implementation of the underlying lighting service.
Jini extends RMI (Chapter 13) to provide services, such as those mentioned above, to a network. Jini services are plug-and-play—clients can discover services on the network dynamically, transparently download classes required to use those services, then begin interacting with those services. Jini, like RMI, requires clients to know the interface of a service to use that service. However, RMI’s dynamic class-downloading capability enables Jini clients to use services without installing special driver software for those services in advance. For Jini clients to discover and use Jini services, standardized interfaces for common services must be developed. Many such interfaces are under development now. For example, printer manufacturers are working on a standard interface for Jini-based printing services.

In this chapter, we introduce Jini technology to build plug-and-play, network-based services. We begin with Jini lookup services, which enable clients to discover other services on the network. We build a simple Jini service that provides information about fictitious professional seminars offered by Deitel & Associates, Inc. and we build a simple Jini client to use that service. The chapter ends with introductions to various Jini utility classes that facilitate discovering lookup services, registering Jini services and building Jini clients. After completing this chapter, you will be able to build simple Jini services and clients that interact with those services. [Note: The commands to execute the examples in this chapter often are quite long. For this reason, we provide batch files containing the commands. These batch files can be found with the examples on the CD that accompanies this book. You can modify these batch files as specified in the text, then use them to execute the programs.]

A new technology that is generating excitement in the computing industry is peer-to-peer computing, which enables communication between computers without connecting through a centralized server. In Chapter 28, we use the Jini technology presented in this chapter as one means of implementing a peer-to-peer instant-messaging application.

### 22.2 Installing Jini

The basic software requirements for Jini include the Java 2 Standard Edition (J2SE) and the *Jini Technology Starter Kit*. If you are going to write commercial Jini services and want to test their compatibility with the Jini platform, you also need to download the *Jini Technology Core Platform Compatibility Kit* (Jini TCK).

The Jini Starter Kit has three components—the *Jini Technology Core Platform* (JCP), the *Jini Technology Extended Platform* (JXP) and the *Jini Software Kit* (JSK). The JCP contains the fundamental Jini interfaces and classes. The JXP provides helper utilities for implementing Jini services and clients. The JSK contains an implementation of the services specified in the JCP and the JXP. The JSK also includes a *JavaSpaces* technology implementation. We discuss JavaSpaces in Chapter 23.

Jini can be downloaded from Sun’s Web site at:


Downloading Jini requires a free registration. After completing the registration, download and extract `JINI-1.1-G-CS.zip`, which contains the Jini Technology Starter Kit. At the time of this writing, the latest version of Jini is 1.1.

### 22.3 Configuring the Jini Runtime Environment

To compile and execute Jini services and clients the JAR files *jini-core.jar*, *jini-ext.jar* and *sun-util.jar* must be included in the `CLASSPATH` environment vari-
able. These three JAR files are in the lib directory of the Jini Starter Kit—they correspond to the Jini Technology Core Platform, the Jini Technology Extended Platform and the Jini Software Kit, respectively. To set the CLASSPATH environment variable on Windows platforms, enter

```
set CLASSPATH=C:\jini1_1\lib\jini-core.jar;C:\jini1_1\lib\jini-ext.jar;C:\jini1_1\lib\sun-util.jar;%CLASSPATH%;.
```

in a command window. To set the CLASSPATH environment variable on most UNIX systems, enter

```
set CLASSPATH=/jini1_1/lib/jini-core.jar:/jini1_1/lib/jini-ext.jar:/jini1_1/lib/sun-util.jar:$CLASSPATH:.
export CLASSPATH
```

in a command window. Note that you will need to do this in every command window from which you intend to use Jini unless you set the CLASSPATH permanently on your system. See your operating system’s documentation for information on setting environment variables. These are the only Jini JAR files that should be in the CLASSPATH. Be sure not to include other JAR files from the Jini distribution. If you extracted the Jini files to a different directory, be sure to specify the appropriate directory name.

### 22.4 Starting the Required Services

Jini depends heavily on a number of network services at run time. Jini uses Java and RMI to provide facilities for downloading resources from the network and for moving Java objects from one JVM to another on demand. To achieve these capabilities, Jini provides various support services. The Jini distribution comes with three services that must be running correctly before executing Jini applications. The following required services must be started:

1. A Web server to enable Jini clients to download class files through RMI, so the clients can access Jini services dynamically.
2. The RMI activation daemon (rmid) to enable the RMI infrastructure that allows Jini clients to communicate with Jini services. The RMI activation daemon enables Activatable services to function properly.
3. A lookup service to maintain information about available Jini services, and to enable clients to discover and use those services.

**Testing and Debugging Tip 22.1**

*The Web server and rmid must be executing (they can be started in any order) before starting the lookup service.*

The Jini Technology Core Platform implementation includes the StartService GUI tool for launching required services. Before running the StartService tool, be sure to set the CLASSPATH environment variable properly as specified in Section 22.3. To start the GUI on Windows, enter the following at a command prompt:

```
java -classpath %CLASSPATH%;C:\jini1_1\lib\jini-examples.jar com.sun.jini.example.launcher.StartService
```
Figure 22.1 shows the **StartService** user interface. Jini provides a basic *property file* for configuring the **StartService** tool. For Windows, the property file is `jini1_1\example\launcher\jini11_win32.properties`. To load this property file, go to the **File** menu, select **Open Property File** and select the property file from the appropriate directory.

**Configure the Web Server**

To configure the Web server, select the **WebServer** panel. Figure 22.2 shows the values for configuring the Web server. If you install the Jini Starter Kit in a directory other than `C:\jini1_1`, be sure to enter the appropriate directory name. Note the **Port** number in Fig. 22.2, which is required in each of the later examples.

---

Fig. 22.1  **StartService** window.

---

Fig. 22.2  **WebServer** configuration tab.
Configure the RMI Activation Daemon

To configure the RMI activation daemon (`rmid`), select the RMID panel. Figure 22.3 shows the default values used to configure the RMI activation daemon. If you want to use your own policy, or you want to add other options (i.e., port number or log directory), you can put all of them in the Options parameter. Use a single space to separate the options. Figure 22.4 shows a sample that specifies the directory in which `rmid` will write its log files.

Configure the Lookup Service

To configure the lookup service, select the Reggie panel. Figure 22.5 shows the configuration values. Replace hostname in the Codebase field with the name or IP address of your computer. Replace the port number (8081) with the port number you specified in the Web server’s configuration (e.g., 9000). Also, be sure to specify for Log Directory a directory that actually exists on your computer, or create a new directory, to avoid exceptions when running the lookup service. The Groups value in this example (public) specifies that the lookup service supports any clients on the network. You can define other values that can be used to restrict the set of clients supported by the lookup service.
Common Programming Error 22.1
Specifying the local computer’s name or using a file URL for the Codebase is an error. Use an http URL with a fully qualified name or IP address.

Common Programming Error 22.2
Starting the lookup service without creating a log directory causes exceptions at run-time. Create the log directory before starting the lookup service.

Common Programming Error 22.3
Create different log directories if you run multiple lookup service instances. Failure to do so can cause an exception that says the directory exists when trying to start a lookup service.

Starting Required Services
To run the required services, select the Run panel in the StartService window (Fig. 22.6). Click the Start RMID button to start the RMI activation daemon. Click the Start WebServer button to start the Web server. Click the Start Reggie button to start the lookup service. Remember to start the RMI activation daemon and the Web server before starting the lookup service.
22.5 Running the Jini LookupBrowser

After starting these services, use the Jini LookupBrowser to test your configuration. The following command starts the browser on Windows:

```java
java -cp c:\jini1_1\lib\jni-examples.jar
    -Djava.security.policy=c:\jini1_1\example\browser\policy
    -Djava.rmi.server.codebase=http://hostname:port/
    jini-examples-dl.jar
    com.sun.jini.example.browser.Browser
```

where `hostname` is the hostname running the Web server and `port` is the port number for the Web server (e.g., 9000).

You also can start the browser via the StartService GUI. Select the LookupBrowser panel. Replace `hostname` with the name or IP address of the computer on which the Web server is running. Change the port number if you specified a different one when you started the Web server. Then go to the Run panel and click the Start LookupBrowser button. Figure 22.7 shows the LookupBrowser panel in the StartService window.

Figure 22.8 shows the result of running the LookupBrowser sample. This output indicates that there is one lookup service registered with `rmid`. Actually, this uses a discovery protocol to find the lookup service. If you set up two lookup services, you would see “2 registrars, not selected.” The number of registrars found is equal to the number of lookup services. If you see “no registrar to select,” the part of the prior configuration was performed incorrectly. In this case, check whether the lookup service started, then check that you specified the correct Codebase in the Reggie panel.

If you click the Registrar menu, you should see the `hostname` (or IP address) and port number on which the lookup service registered. In Fig. 22.9, the `hostname` is DRAGONFLY. The default port number was used, so no port number appears in this screen capture.
22.6 Discovery

The Jini lookup service is the heart of a Jini community. The process of finding the lookup services and obtaining references to them is called discovery. A service registers itself with one or more lookup services to make itself available to clients. To do so, the services must first discover the lookup services. Clients search the lookup services to locate the services they require. To do so, the clients must first discover the lookup services. Hence discovery is a common task for both services and clients.

Discovery distinguishes Jini technology from RMI. In RMI, you must know in advance where to register an object. In Jini, you do not need to know where—just how. The discovery process determines where, but hides the details from the developer. Discovery can be accomplished using either unicast discovery or multicast discovery.

22.6.1 Unicast Discovery

Unicast discovery, or locator discovery, enables a Jini service or client to discover lookup services on a specific host. The Jini service or client sends a discovery request to the computer, which responds with a remote reference to the lookup service running on that computer on the given port.

The application of Fig. 22.10 demonstrates unicast discovery. Class UnicastDiscovery uses class net.jini.core.discovery.LookupLocator to perform unicast discovery. Lines 17–18 import class LookupLocator, for discovering lookup services, and interface ServiceRegistrar (package net.jini.core.lookup), which represents a lookup service. The UnicastDiscovery constructor (lines 29–52) creates JButton discoverButton and JTextArea outputArea. When the user clicks discoverButton, line 43 invokes method discoverLookupServices, which displays information about discovered lookup services in outputArea.
// UnicastDiscovery.java
// UnicastDiscovery is an application that demonstrates Jini
// lookup service discovery for a known host (unicast).
package com.deitel.advjhtp1.jini.discovery;

import java.rmi.*;
import java.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;

public class UnicastDiscovery extends JFrame {
    private JTextArea outputArea = new JTextArea(10, 20);
    private JButton discoverButton;

    // hostname for discovering lookup services
    private String hostname;

    // UnicastDiscovery constructor
    public UnicastDiscovery(String host) {
        super("UnicastDiscovery Output");
        hostname = host; // set target hostname for discovery
        discoverButton = new JButton("Discover Lookup Services");
        discoverButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                discoverLookupServices();
            }
        });
        Container contentPane = getContentPane();
        contentPane.add(outputArea, BorderLayout.CENTER);
        contentPane.add(discoverButton, BorderLayout.NORTH);
    }
}

Fig. 22.10 UnicastDiscovery performs unicast discovery to locate Jini lookup services (part 1 of 3).
// discover lookup services on given host and get details
// about each lookup service from ServiceRegistrar
public void discoverLookupServices()
{
    // construct Jini URL
    String lookupURL = "jini://" + hostname + "/";

    // connect to the lookup service at lookupURL
    try {
        LookupLocator locator = new LookupLocator( lookupURL );
        outputArea.append( "Connecting to " + lookupURL + "\n" );

        // perform unicast discovery to get ServiceRegistrar
        ServiceRegistrar registrar =
            locator.getRegistrar();

        // print lookup service information and
        outputArea.append( "Got ServiceRegistrar\n" +
            "  Lookup Service Host: " + locator.getHost() + "\n" +
            "  Lookup Service Port: " + locator.getPort() + "\n" );

        // get groups that lookup service supports
        String[] groups = registrar.getGroups();
        outputArea.append( "Lookup service supports " +
            " group(s):\n" );

        // get group names; if empty, write public
        for ( int i = 0; i < groups.length ; i++ ) {
            if ( groups[ i ].equals( "" ) )
                outputArea.append( "  public\n" );
            else
                outputArea.append( "  " + groups[ i ] + "\n" );
        }
    }

    // handle exception if URL is invalid
    catch ( MalformedURLException exception ) {
        exception.printStackTrace();
        outputArea.append( exception.getMessage() );
    }

    // handle exception communicating with ServiceRegistrar
    catch ( RemoteException exception ) {
        exception.printStackTrace();
        outputArea.append( exception.getMessage() );
    }
}

Fig. 22.10 UnicastDiscovery performs unicast discovery to locate Jini lookup services (part 2 of 3).
Method `discoverLookupServices` (lines 56–115) locates services on a particular computer. Line 59 creates a `String` that represents the URL for the computer running the lookup services. This URL must specify the `jini` protocol, a hostname and, optionally, the port on which to connect (e.g., `jini://mycomputer.mydomain.com:1234`). If the URL does not specify a port, the default port `4160` is used. Class `UnicastDiscovery` reads the hostname from a command-line argument and stores it in instance variable `hostname`. Line 63 creates a new `LookupLocator` for discovering lookup services. This `LookupLocator` constructor takes as an argument the Jini URL for the
computer running the lookup service. A `MalformedURLException` occurs if the URL specified does not follow the appropriate format.

Line 68 invokes method `getRegistrar` of class `LookupLocator` to perform unicast discovery. Method `getRegistrar` returns a `ServiceRegistrar`, which represents a lookup service. An overloaded version of method `getRegistrar` takes as an integer argument the maximum number of milliseconds to wait for the unicast discovery to locate a `ServiceRegistrar` before issuing a timeout.

Lines 71–88 display information about the discovered lookup service in output-Area. Lines 72–73 invoke methods `getHost` and `getPort` of class `LookupLocator` to retrieve the hostname and port number where the lookup service was discovered. Line 76 invokes method `getGroups` of interface `ServiceRegistrar` to retrieve an array of supported group names. An empty string indicates the public group.

Method `main` (lines 118–140) launches the `UnicastDiscovery` application. Lines 121–122 install an `RMISecurityManager` to enable network class loading—downloading executable code over the network. When class `UnicastDiscovery` discovers a new lookup service, the classes that implement that lookup service are loaded over the network. This class loading is a security risk, because the code will execute locally. A malicious programmer could access, modify or destroy sensitive data on the local machine. Installing a `SecurityManager` restricts downloaded code from performing tasks that are not granted explicitly in the current security policy. For more information on setting security policies, please see Chapter 7, Security. For the examples in this chapter, we use the policy file of Fig. 22.11, which grants `AllPermission` to the code in each example. This is a security risk and should not be used with production applications.

Lines 126–128 of Fig. 22.10 check the command-line arguments for a hostname and print out usage instructions if no hostname is provided. Lines 133–138 create a new `UnicastDiscovery` instance and display the user interface.

Before executing class `UnicastDiscovery`, start `rmid`, the Web server and the Reggie lookup service (if necessary, review Section 22.4). To execute the `UnicastDiscovery` application, enter

```
java -Djava.security.policy=policy
      com.deitel.advjhtp1.jini.discovery.UnicastDiscovery host
```

where `policy` is a security file that specifies the security policy and `host` is the hostname or IP address of the computer running the lookup service. Be sure that you execute this command from the directory that contains the `com.deitel.advjhtp1` package structure, and be sure that the `CLASSPATH` environment variable includes the current directory. Figure 22.12 shows the running application.

```java
// policy.all
// grant AllPermission to all code (DANGEROUS!)
grant {
  permission java.security.AllPermission "", "";
};
```

Fig. 22.11 Policy file that grants `AllPermission` to all code.
22.6.2 Multicast Discovery

Multicast discovery, or group discovery, enables a Jini service or client to discover lookup services when the particular host running the lookup service is not known. Recall that when using unicast discovery, the Jini service or client must request lookup services from a particular host. A multicast discovery request uses network multicast to discover nearby lookup services. Lookup services, in turn, periodically issue multicast announcements to notify interested Jini services and clients that the lookup services are available.

The application of Fig. 22.13 demonstrates multicast discovery. Class `MulticastDiscovery` uses class `net.jini.discovery.LookupDiscovery` to perform discovery. Class `MulticastDiscovery` implements the interface `DiscoveryListener` (line 22) to enable the class `MulticastDiscovery` to receive `DiscoveryEvent`s—notifications of discovered lookup services. Line 36 creates a `LookupDiscovery`. The `LookupDiscovery` constructor takes as an argument an array of `Strings` in which each element is the name of a group. The `LookupDiscovery` object will discover all nearby lookup services that support the groups specified in this `String` array. Line 37 creates a new `String` array with an empty string as its only element. This indicates that the `LookupDiscovery` instance should discover lookup services that support the “public” group. Line 43 invokes method `addDiscoveryListener` of class `LookupDiscovery` to register the `MulticastDiscovery` object as the listener for `DiscoveryEvent`s.

Class `LookupDiscovery` invokes method `discovered` (lines 56–114) when `LookupDiscovery` locates new lookup services. Line 59 invokes method `getRegistrars` of class `DiscoveryEvent` to obtain an array of discovered `ServiceRegistrars`. Lines 74–80 create a new `TextAppender` that contains information about a `discoveredServiceRegistrar`. Class `TextAppender` (lines 130–148) appends text to the `outputArea`. Lines 75–80 obtain information about the `ServiceRegistrar` and build an output string. Line 83 invokes method `invokeLater` of class `SwingUtilities` with the `TextAppender` as an argument to append the text to `outputArea`. Lines 91–97 obtain group information from the `ServiceRegistrar` and lines 100–101 append the group information to `outputArea`.

Class `LookupDiscovery` invokes method `discarded` (lines 118–126) when a lookup service should be discarded because it is no longer available or because it no longer matches the set of groups in which the Jini service or client is interested. Lines 120–121 invoke method `getRegistrars` of class `DiscoveryEvent` to obtain an array of discarded `ServiceRegistrars`. Lines 123–125 add the number of discarded `ServiceRegistrars` to the display.
MulticastDiscovery.java
MulticastDiscovery is an application that demonstrates Jini lookup service discovery using multicast.

```
package com.deitel.advjhtp1.jini.discovery;

// Java core packages
import java.rmi.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

// Jini core packages
import net.jini.core.lookup.ServiceRegistrar;

// Jini extension packages
import net.jini.discovery.*;

public class MulticastDiscovery extends JFrame
  implements DiscoveryListener {

  // number of lookup services discovered through multicast
  private int servicesFound = 0;

  private JTextArea outputArea = new JTextArea( 10, 20 );

  // MulticastDiscovery constructor
  public MulticastDiscovery ()
  {
    super( "MulticastDiscovery" );

    // discover lookup services in public group using multicast
    try {
      LookupDiscovery lookupDiscovery = new LookupDiscovery( new String[] { "" } );

      outputArea.append( "Finding lookup services for " +
                        "public group ...
                      " );

      // listen for DiscoveryEvents
      lookupDiscovery.addDiscoveryListener ( this );
    }

    // handle exception discovering lookup services
    catch ( IOException exception ) {
      exception.printStackTrace();
    }
}
```

Fig. 22.13 MulticastDiscovery performs multicast discovery to locate Jini lookup services (part 1 of 4).
```java
getContentPane().add( new JScrollPane( outputArea ), BorderLayout.CENTER );
}

// receive notification of found lookup services
public void discovered ( DiscoveryEvent event ) {
    // get the ServiceRegistrars for found lookup services
    ServiceRegistrar[] registrars = event.getRegistrars();
    int order = 0;

    // get the information for each lookup service found
    for ( int i = 0; i < registrars.length ; i++ ) {
        ServiceRegistrar registrar = registrars[ i ];

        if ( registrar != null ) {
            // append information about discovered services
            // to outputArea
            try {
                order = servicesFound + i + 1;

                // get the hostname and port number
                Runnable appender = new TextAppender(
                        "Lookup Service " + order + ":\n" +
                        " Host: " +
                        registrar.getLocator().getHost() + "\n" +
                        "\n Port: " +
                        registrar.getLocator().getPort() + "\n" +
                        " Group support: " );

                // append to outputArea on event-dispatch thread
                SwingUtilities.invokeLater( appender );

                // get the group(s) the lookup service served
                String[] groups = registrar.getGroups();

                StringBuffer names = new StringBuffer();

                // get the group names, if empty write public
                for ( int j = 0; j < groups.length ; j++ ) {
                    if ( groups[ j ].equals( "" ) )
                        names.append( "public\t" );
                    else
                        names.append( groups[ j ] + "\t" );
                }

                // append group names to outputArea
                SwingUtilities.invokeLater( new TextAppender( names + "\n" ) );
            }
        }
    }
}
```

---

Fig. 22.13 MulticastDiscovery performs multicast discovery to locate Jini lookup services (part 2 of 4).
// handle exception communicating with ServiceRegistrar

try {
    exception.printStackTrace();
} catch (RemoteException exception) {
}

// update number of services found
servicesFound = order;

} // end method discovered

// receive notification of discarded lookup services that
// are no longer valid
public void discarded(DiscoveryEvent event) {
    ServiceRegistrar[] discardedRegistrars = event.getRegistrars();

    SwingUtilities.invokeLater(
        new TextAppender("Number of discarded registrars: " +
                        discardedRegistrars.length + "\n")
    );
}

// TextAppender is a Runnable class for appending
// text to outputArea on the event-dispatching thread.
private class TextAppender implements Runnable {

    private String textToAppend; // text to append to outputArea

    // TextAppender constructor
    public TextAppender(String text) {
        textToAppend = text;
    }

    // append text to outputArea and scroll to bottom
    public void run() {
        outputArea.append(textToAppend);
        outputArea.setCaretPosition(outputArea.getText().length());
    }

} // end inner class TextAppender

// launch MulticastDiscovery application
public static void main(String args[]) {

Fig. 22.13 MulticastDiscovery performs multicast discovery to locate Jini lookup services (part 3 of 4).
Before running this example, start several instances of the Reggie lookup service. If there is only one lookup service running, the result is the same as the Unicast-Discovery example (Fig. 22.10). To start multiple instances of the Reggie lookup service, specify a different log directory in the Reggie panel, then click Start Reggie from the Run panel. Try adding new group names to some of the lookup services (e.g., “test” or “MyGroup”). To execute MulticastDiscovery, enter the following at a command prompt:

```java
java -Djava.security.policy=.policy
com.deitel.advjhtp1.jini.discovery.MulticastDiscovery
```

where policy is an appropriate security policy. Figure 22.14 shows the output window with several lookup services running.
22.7 Jini Service and Client Implementations

In this section, we develop a Jini service that provides information about fictitious seminars offered at Deitel & Associates, Inc. Then, we create a client for that service. A Jini service consists of several components, each of which contributes to the flexibility and portability of the Jini architecture. A service proxy is an intermediary between a Jini service and its clients. The seminar service proxy implements the service’s public interface, which declares the methods that the service provides. The service proxy communicates with the actual service implementation through the service’s back-end interface, which defines methods in the service implementation. A separate application discovers lookup services and registers the Jini service, making the service available to Jini clients.

**Software Engineering Observation 22.1**

Providing a back-end interface for the service implementation is not required. However, using a back-end interface makes the Jini service more flexible, since the back-end implementation can be changed without requiring changes in the service proxy.

**Software Engineering Observation 22.2**

An alternative to providing a back-end interface and implementation is to implement the service’s functionality in the service proxy itself.

A Jini client uses the lookup service discovery techniques presented earlier in this chapter to discover lookup services. The Jini client then uses the discovered lookup services to locate the desired Jini service. When the lookup service locates the service requested by the Jini client, the lookup service serializes the service proxy and delivers the proxy to the Jini client. The client can then invoke methods defined in the service’s public interface directly on the service proxy, which implements that interface. The service proxy communicates with the service implementation through the back-end interface.

Our Jini service provides information about fictitious seminars offered at Deitel & Associates, Inc. Information about these seminars is stored in instances of class Seminar (Fig. 22.15). Interface SeminarInterface (Fig. 22.16) is the public interface for the Jini service. Class SeminarProxy (Fig. 22.18) implements interface SeminarInterface and communicates with the service implementation through BackendInterface (Fig. 22.17). Class SeminarInfo (Fig. 22.19) is the service implementation, which implements interface BackendInterface. Class SeminarInfoService (Fig. 22.21) discovers lookup services and registers the SeminarInfo Jini service. Class UnicastSeminarInfoClient (Fig. 22.22) is a Jini client that uses unicast discovery to discover lookup services and locate the SeminarInfo Jini service. This client allows a user to select a day of the week and view the seminars offered on that day.

### 22.7.1 Service Interfaces and Supporting Classes

Class Seminar (Fig. 22.15) represents a fictitious seminar, including the seminar’s title and location. It implements Serializable, so that objects of this class can be sent from the Jini service to its clients across the network.

Line 13 explicitly specifies the serialVersionUID for class Seminar. Developers can define this static member in Serializable classes to ensure compatibility between versions of such classes. If an object of one version of a class is serialized, the object can be deserialized into an object of a newer version of the class as long as both ver-
vions use the same `serialVersionUID` (and are implemented in a compatible manner). Changing the `serialVersionUID` for a new version of the class indicates that the new version is not compatible with older versions. In this case, deserialization would not work correctly.

Interface `SeminarInterface` (Fig. 22.16) defines a single method `getSeminar`, which takes as a `String` argument a day of the week. Method `getSeminar` returns a `Seminar` object containing information about the `Seminar` offered on the given day. The service proxy must implement this interface, because Jini clients use this interface to interact with the service.

```java
package com.deitel.advjhtp1.jini.seminar;

// Seminar.java
// Seminar represents a seminar, or lecture, including the
// Seminar title and location.
public class Seminar implements Serializable {

    private String title;
    private String location;
    private static final long serialVersionUID = 20010724L;

    // Seminar constructor
    public Seminar(String seminarTitle, String seminarLocation) {
        title = seminarTitle;
        location = seminarLocation;
    }

    // get String representation of Seminar object
    public String toString() {
        return "Seminar title: " + getTitle() + "; location: " + getLocation();
    }

    // get Seminar title
    public String getTitle() {
        return title;
    }

    // get Seminar location
    public String getLocation() {
        return location;
    }
}
```

Fig. 22.15 `Seminar` maintains the location and title of a seminar.
Interface BackendInterface (Fig. 22.17) defines methods that the service proxy uses to communicate with the service implementation. The proxy invokes method \texttt{getSeminar} to retrieve Seminar information for the requested day. In this Jini service, the seminar proxy communicates with the service implementation using RMI, although Jini implementations may use any protocol to communicate with back-end implementations.

Software Engineering Observation 22.3

\textit{Jini does not require that service proxies communicate with back-end implementations using RMI. Service proxies may use RMI, TCP/IP, CORBA or any suitable protocol to connect to back-end implementations.}

Fig. 22.17 BackendInterface defines methods available to the SeminarInfo service proxy.
Software Engineering Observation 22.4

Service proxies can use non-Java protocols (e.g., TCP/IP or CORBA) to communicate with back-end implementations, enabling programmers to provide Jini services that are implemented in languages other than Java.

22.7.2 Service Proxy and Service Implementations

Class SeminarProxy (Fig. 22.18) is a service proxy for the SeminarInfo Jini service. Line 12 specifies that class SeminarProxy implements SeminarInterface, which is the public interface for the SeminarInfo Jini service. The service proxy must implement this interface to enable Jini clients to communicate with the SeminarInfo service. The service proxy also must implement interface Serializable (line 13), so that instances can be delivered to the remote client over RMI. The SeminarProxy constructor (lines 18–21) initializes the proxy with a remote reference to the back-end implementation. Method getSeminar (lines 24–38) invokes the back-end implementation’s getSeminar method to retrieve the seminar information (line 28).

```java
// SeminarProxy.java
// SeminarProxy is a proxy for the SeminarInfo Jini service.
package com.deitel.advjhtp1.jini.seminar.service;

// Java core packages
import java.io.Serializable;
import java.rmi.*;

// Deitel packages
import com.deitel.advjhtp1.jini.seminar.Seminar;

public class SeminarProxy implements SeminarInterface, Serializable {
    private BackendInterface backInterface;

    // SeminarProxy constructor
    public SeminarProxy( BackendInterface inputInterface ) {
        backInterface = inputInterface;
    }

    // get Seminar for given date through BackendInterface
    public Seminar getSeminar( String date ) {
        // get Seminar from service through BackendInterface
        try {
            return backInterface.getSeminar( date );
        }
    }
}
```

Fig. 22.18 SeminarProxy is a service proxy that clients use to communicate with the SeminarInfo service (part 1 of 2).
Class **SeminarInfo** (Fig. 22.19) is an RMI object that implements the Jini service’s back-end interface. The service proxy communicates with this back-end implementation through interface **BackendInterface** using RMI. Method **getSeminar** (lines 29–91) reads seminar information from file **SeminarInfo.txt** and returns a new **Seminar** object containing seminar information for the given day of the week.

File **SeminarInfo.txt** (Fig. 22.20) contains seminar information that the **SeminarInfo** service provides to its clients. A semicolon separates the **Seminar** title from the **Seminar** location.
public Seminar getSeminar( String date )
    throws RemoteException
{
    String[] titles = new String[] { "", "", "", "", "" }; // get Seminar information for given day
    String[] locations = new String[] { "", "", "", "", "" };

    // read seminar information from text file
    try {
        String fileName = SeminarInfo.class.getResource("SeminarInfo.txt").toString();
        fileName = fileName.substring( 6 );

        FileInputStream inputStream =
            new FileInputStream( fileName );

        BufferedReader reader = new BufferedReader(
            new InputStreamReader( inputStream ));

        String line = reader.readLine();

        // read seminar info from the file
        for ( int lineNo = 0; ( line != null )
                && ( lineNo < 5 ); lineNo++ ) {
            StringTokenizer tokenizer =
                new StringTokenizer( line, ";" );

            titles[ lineNo ] = tokenizer.nextToken();
            locations[ lineNo ] = tokenizer.nextToken();
            line = reader.readLine();
        }
    }

    // handle exception loading Seminar file
    catch ( FileNotFoundException fileException ) {
        fileException.printStackTrace();
    }

    // handle exception reading from Seminar file
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }

    // match given day of the week to available seminars
    if ( date.equalsIgnoreCase( MONDAY ) ) {
        return new Seminar( titles[ 0 ], locations[ 0 ] );
    }
    else if ( date.equalsIgnoreCase( TUESDAY ) ) {
        return new Seminar( titles[ 1 ], locations[ 1 ] );
    }
    else if ( date.equalsIgnoreCase( WEDNESDAY ) ) {
        return new Seminar( titles[ 2 ], locations[ 2 ] );
    }
}

Fig. 22.19 SeminarInfo implements the SeminarInfo Jini service (part 2 of 3).
22.7.3 Registering the Service with Lookup Services

Class SeminarInfoService (Fig. 22.21) discovers lookup services using multicast discovery and registers the SeminarInfo service with discovered lookup services. Lines 29–30 create a LookupDiscovery object to perform multicast discovery for the “public” group. Line 33 registers the SeminarInfoService object as a DiscoveryListener to receive lookup service discovery notifications. Line 42 creates an array of Entry objects. An Entry (in package net.jini.core.entry) describes a service, which enables Jini clients to search for services of a particular description. Line 43 creates a new NameEntry (in package net.jini.lookup.entry) and adds it to array entries to provide the name of the Jini service. Lines 46–47 create a new ServiceItem (in package net.jini.core.lookup) for the SeminarInfo Jini service. The lookup service requires a ServiceItem to register a Jini service. The first argument to the ServiceItem constructor is the Jini service’s ID. The null argument on line 53 causes the lookup service to assign a new, unique ID to the service. To keep the services persistent, the service provider should use the previously assigned service ID when re-registering services. The second argument is an instance of the service proxy for the Jini service. The third argument is the array of Entry objects that describe the service.

Fig. 22.19 SeminarInfo implements the SeminarInfo Jini service (part 3 of 3).

Fig. 22.20 Content of SeminarInfo.txt.

Fig. 22.21 SeminarInfoService registers the SeminarInfo service with lookup services (part 1 of 4).
```java
// Java core packages
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.io.IOException;

// Jini core packages
import net.jini.core.lookup.*;
import net.jini.core.entry.Entry;

// Jini extension packages
import net.jini.discovery.*;
import net.jini.lookup.entry.Name;

public class SeminarInfoService implements DiscoveryListener {

    private ServiceItem serviceItem;
    private final int LEASETIME = 10 * 60 * 1000;

    // SeminarInfoService constructor
    public SeminarInfoService() {
        // search for lookup services with public group
        try {
            LookupDiscovery discover =
                new LookupDiscovery( new String[] { "" } );

            // add listener for DiscoveryEvents
            discover.addDiscoveryListener( this );
        }

        // handle exception creating LookupDiscovery
        catch ( IOException exception ) {
            exception.printStackTrace();
        }

        // create an Entry for this service
        Entry[] entries = new Entry[ 1 ];
        entries[ 0 ] = new Name( "Seminar" );

        // set the service's proxy and Entry name
        serviceItem = new ServiceItem( null, createProxy(), entries );
    }

    // receive lookup service discovery notifications
    public void discovered( DiscoveryEvent event ) {
        ServiceRegistrar[] registrars = event.getRegistrars();
```
// register service with each lookup service
for ( int i = 0; i < registrars.length; i++ ) {
    ServiceRegistrar registrar = registrars[ i ];

    // register service with discovered lookup service
    try {
        ServiceRegistration registration =
            registrar.register( serviceItem, LEASETIME );
    }

    // catch the remote exception
    catch ( RemoteException exception) {
        exception.printStackTrace();
    }
}

} // end for

} // end method discovered

// ignore discarded lookup services
public void discarded( DiscoveryEvent event ) {}

// create the seminar service proxy
private SeminarInterface createProxy()
{
    // get BackendInterface for service and create SeminarProxy
    try {
        return new SeminarProxy( new SeminarInfo() );
    }

    // handle exception creating SeminarProxy
    catch ( RemoteException exception ) {
        exception.printStackTrace();
    }

    return null;
}

} // end method discovered

// method main keeps the application alive
public static void main( String args[] )
{
    // set SecurityManager
    if ( System.getSecurityManager() == null )
        System.setSecurityManager( new RMISecurityManager() );

    new SeminarInfoService();

    Object keepAlive = new Object();

Fig. 22.21 SeminarInfoService registers the SeminarInfo service with lookup services (part 3 of 4).
Method discovered (lines 52–73) receives notifications of discovered lookup services. For each discovered lookup service, line 62 invokes method register of interface ServiceRegistrar to register the Jini service’s serviceItem with the lookup service. Method register takes as arguments a serviceItem, which contains the service proxy and a lease time. The lease time specifies the length of time for which the service should be available from the lookup service. For this example, the lease time is 10 minutes. The LEASETIME specified in method register is the service requested lease time. In fact, Sun’s implementation limits it to 5 minutes. This means the desired lease time is not necessarily granted. After 10 minutes or even less, the Jini service’s lease expires and the service is no longer available through the lookup service. It is the service provider’s responsibility to store the ServiceRegistration (line 62) returned by method register and use it to periodically renew the lease. We discuss leasing and how to manage leases in Section 22.8.3.

Method createProxy (lines 79–93) creates a new SeminarProxy service proxy for our Jini service. Line 83 creates a new instance of the SeminarInfo back-end implementation. Line 84 returns a new SeminarProxy service proxy for the SeminarInfo service back-end implementation.

Method main (lines 95–118) launches the SeminarInfoService application. Line 99 sets a SecurityManager and line 101 creates a new instance of class SeminarInfoService, which discovers lookup services and registers the SeminarInfo service. Lines 103–115 create an Object named keepAlive. The synchronized block (lines 105–116) prevents the main thread from terminating, which would cause the SeminarInfoService application to terminate, thus shutting down the SeminarInfo service. Clients then would be unable to access Seminar information.

22.7.4 Jini Service Client

Class UnicastSeminarInfoClient (Fig. 22.22) is a Jini client that uses the SeminarInfo service to retrieve information about Seminars available on a given day. Class UnicastSeminarInfoClient performs unicast discovery to discover lookup
services. The **UnicastSeminarInfoClient** constructor (lines 44–82) initializes the application GUI.

Line 51 creates a **JButton** that the user can click to begin the discovery process and retrieve **Seminar** information. Line 61 invokes method **discoverLookupServices** to discover lookup services using unicast discovery. Lines 63–64 obtain a remote reference to the **SeminarInfo** service’s service proxy by invoking method **lookupSeminarService**. Lines 66–69 prompt the user for the day of the week for which the user would like **Seminar** information. Line 71 invokes method **showSeminars** to display the **Seminar** information for the selected day.

---

```java
// UnicastSeminarInfoClient.java
// UnicastSeminarInfoClient uses unicast discovery to locate lookup services for the SeminarInfo service.
package com.deitel.advjhtp1.jini.seminar.client;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.rmi.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

// Jini core packages
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.*;
import net.jini.core.entry.Entry;

// Jini extension packages
import net.jini.lookup.entry.Name;

// Deitel packages
import com.deitel.advjhtp1.jini.seminar.Seminar;
import com.deitel.advjhtp1.jini.seminar.service.SeminarInterface;

public class UnicastSeminarInfoClient extends JFrame {

    // Strings representing the days of the week on which Seminars are offered
    private static final String[] days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};

    // hostname and ServiceRegistrar for lookup services
    private String hostname;
    private ServiceRegistrar registrar;
```

**Fig. 22.22 UnicastSeminarInfoClient** is a client for the SeminarInfo service (part 1 of 4).
```java
    // JButton for finding Seminars 
    private JButton findSeminarButton; 

    // UnicastSeminarInfoClient constructor 
    public UnicastSeminarInfoClient( String host ) 
    { 
      super( "UnicastSeminarInfoClient" ); 

      hostname = host; // set target hostname for discovery 

      // create JButton for finding Seminars 
      findSeminarButton = new JButton( "Find Seminar" ); 
      findSeminarButton.addActionListener( 

        new ActionListener() { 

          // discover lookup services, look up SeminarInfo 
          // service, prompt user for desired day of the 
          // week and display available Seminars 
          public void actionPerformed( ActionEvent event ) 
          { 
            discoverLookupServices(); 

            SeminarInterface seminarService = 
              lookupSeminarService(); 

            String day = ( String ) JOptionPane.showInputDialog( 
              UnicastSeminarInfoClient.this, "Select Day", 
              "Day Selection", JOptionPane.QUESTION_MESSAGE, 
              null, days, days[ 0 ] ); 

            showSeminars( seminarService, day ); 
          } 
        } ); // end call to addActionListener 

      JPanel buttonPanel = new JPanel(); 
      buttonPanel.add( findSeminarButton ); 

      getContentPane().add( buttonPanel, BorderLayout.CENTER ); 
    } // end UnicastSeminarInfoClient constructor 

    // discover lookup services using unicast discovery 
    private void discoverLookupServices() 
    { 
      String lookupURL = "jini://" + hostname + "/"; 

      // Get the lookup service locator at jini://hostname 
      // use default port 
      try { 
```
LookupLocator locator = new LookupLocator( lookupURL );

// return registrar
registrar = locator.getRegistrar();

// handle exceptions discovering lookup services
try {
    registrar = locator.getRegistrar();
} catch ( Exception exception ) {
    exception.printStackTrace();
}

// lookup SeminarInfo service in given ServiceRegistrar
private SeminarInterface lookupSeminarService()
{
    // specify the service requirement
    Class[] types = new Class[] { SeminarInterface.class };
    Entry[] attribute = new Entry[] { new Name( "Seminar" ) };
    ServiceTemplate template =
        new ServiceTemplate( null, types, attribute );

    // find the service
    try {
        SeminarInterface seminarInterface =
            ( SeminarInterface ) registrar.lookup( template );
        return seminarInterface;
    }

    // handle exception looking up SeminarInfo service
    catch ( RemoteException exception ) {
        exception.printStackTrace();
    }

    return null;
}

// show Seminar in given SeminarInfo service for given
// day of the week
private void showSeminars( SeminarInterface seminarService,
                          String day )
{
    StringBuffer buffer = new StringBuffer();

    // get Seminar from SeminarInfo service
    if ( seminarService != null ) {
        Seminar seminar = seminarService.getSeminar( day );

        // get subject and location from Seminar object
        buffer.append( "Seminar information: \n" );
        buffer.append( day + "\n" );
    }

    // show Seminar in given SeminarInfo service for given
    // day of the week
private void showSeminars( SeminarInterface seminarService,
                          String day )
{
    StringBuffer buffer = new StringBuffer();

    // get Seminar from SeminarInfo service
    if ( seminarService != null ) {
        Seminar seminar = seminarService.getSeminar( day );

        // get subject and location from Seminar object
        buffer.append( "Seminar information: \n" );
        buffer.append( day + "\n" );

Fig. 22.22 UnicastSeminarInfoClient is a client for the SeminarInfo service (part 3 of 4).
Method **discoverLookupServices** (lines 85–103) performs unicast discovery to discover lookup services on a particular host, specified at the command line. Line 92 creates a **LookupLocator**, and line 95 invokes method **getRegistrar** of class **LookupLocator** to obtain a **ServiceRegistrar** reference to the lookup service, which is stored in instance variable **registrar**.

Method **lookupSeminarService** uses the **ServiceRegistrar** discovered in method **discoverLookupServices** to obtain a **SeminarProxy** for the **SeminarInfo** service. Line 109 creates an array of **Class** objects and initializes the first element to **SeminarInterface**'s **Class** object. The **ServiceRegistrar** uses this array of **Class** objects to locate the appropriate service proxies for the Jini client. Line 110 creates an array of **Entry** objects. Recall that when we registered the **SeminarInfo** service, we provided an array of **Entry** objects to describe that service. Line 110 populates the
entries array with a single Name Entry. The ServiceRegistrar uses this array of Entry objects to locate the desired service. Lines 111–112 create a ServiceTemplate (package net.jini.core.lookup) that contains the Class and Entry arrays from lines 109–110. The ServiceRegistrar uses this ServiceTemplate to locate a service that matches the given set of Class and Entry objects. The first argument to the ServiceTemplate constructor is the service ID for the Jini service. We pass a null argument to indicate that we do not know the service ID for the desired service.

Lines 116–117 invoke method lookup of interface ServiceRegistrar to perform the lookup and retrieve a SeminarProxy service proxy. Line 117 passes the ServiceTemplate to method lookup. The ServiceRegistrar matches the information in the ServiceTemplate to the ServiceItems registered in the lookup service. For a ServiceItem to match a ServiceTemplate when a ServiceTemplate contains a non-null service ID, the service ID must match that in the ServiceTemplate. Also, the service must be an instance of every class in the Class array in the ServiceTemplate. Finally, the service attributes must match one or more of the attributes in the Entry array in the ServiceTemplate. Line 118 returns a reference to the SeminarProxy returned by method lookup.

Method showSeminars (lines 132–154) takes as arguments a SeminarInterface and a String containing the day of the week for which the user would like Seminar information. Line 139 invokes method getSeminar of interface SeminarInterface and passes as an argument the day. Recall that SeminarInterface is the public interface for the SeminarInfo Jini service and defines all methods available for that service. Line 152 displays the Seminar information in a JOptionPane message dialog.

Method main (lines 157–178) launches the UnicastSeminarInfoClient application. Lines 160–163 ensure that the user enters the hostname on which to perform lookup service discovery. Lines 167–175 set a SecurityManager and start UnicastSeminarInfoClient.

Running the Service and Client
Clients and services typically run on different computers. To mimic this configuration on a single computer, we must separate the class files for the Jini client, the class files for the Jini service and the downloadable Jini service class files. In this example, we package the service’s class files in a JAR file named SeminarService.jar. Figure 22.23 shows the contents of SeminarService.jar.

<table>
<thead>
<tr>
<th>Class File</th>
<th>Directory in SeminarService.jar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seminar.class</td>
<td>com\deitel\advjhtp1\jni\seminar\</td>
</tr>
<tr>
<td>SeminarInterface.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service\</td>
</tr>
<tr>
<td>SeminarProxy.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service\</td>
</tr>
</tbody>
</table>

Fig. 22.23 SeminarService.jar contents (part 1 of 2).
Recall that class \texttt{SeminarInfo} is an RMI remote object from which the \texttt{SeminarProxy} retrieves \texttt{Seminar} information. This requires that we use the RMI stub compiler (\texttt{rmic}) to compile a stub file for class \texttt{SeminarInfo} and place this stub file (\texttt{SeminarInfo\_Stub.class}) in \texttt{SeminarService.jar}.

We place our Jini client’s class files in JAR file \texttt{SeminarClient.jar}. To access the \texttt{SeminarInfo} Jini Service, the client needs the class file for the service’s \texttt{public} interface, as well as any supporting classes that used by methods in the \texttt{public} interface. \texttt{SeminarClient.jar} contains \texttt{Seminar.class} and \texttt{SeminarInterface.class}, which contains the service’s \texttt{public} interface. Figure 22.24 shows the contents of \texttt{SeminarClient.jar}.

When the client requests the \texttt{SeminarInfo} service from a lookup service, the client uses network class loading to load the service proxy class into memory and execute methods on the Jini service. Therefore, we must package the necessary class files for the Jini client to download. The client requires the service proxy’s class file and class files for objects that the service proxy references. Recall that the client interacts with the service proxy through interface \texttt{SeminarInterface}. The client does not know about the service proxy \texttt{SeminarProxy} or its supporting classes. Therefore, the client must download the \texttt{SeminarProxy} class file and its supporting class files at run time using network class loading. The download classes include \texttt{BackendInterface.class} and \texttt{SeminarInfo\_Stub.class}, which the service proxy uses to communicate with the \texttt{SeminarInfo} remote object. We package these files in \texttt{SeminarServiceDownload.jar} (Fig. 22.25) and publish this JAR file on a Web server for the client to download.

After creating these JAR files, start \texttt{rmid}, a Web server and the Reggie lookup service (if necessary, review Section 22.4 for detailed instructions). Configure and start an additional Web server to enable clients to download the service proxy and supporting files. Place \texttt{SeminarServiceDownload.jar} in the \texttt{Document Area} directory specified in this additional Web server’s configuration (e.g., \texttt{C:\Jini\seminar\service}). Figure 22.26 shows a sample Web server configuration that uses port 9090. We specify this port number in the \texttt{java.rmi.server.codebase} command-line property when we launch the \texttt{SeminarInfo} service.
### SeminarClient.jar contents.

<table>
<thead>
<tr>
<th>Class File</th>
<th>Directory in SeminarClient.jar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seminar.class</td>
<td>com\deitel\advjhtpl\jini\seminar\</td>
</tr>
<tr>
<td>SeminarInterface.class</td>
<td>com\deitel\advjhtpl\jini\seminar\service\</td>
</tr>
<tr>
<td>UnicastSeminarInfoClient.class</td>
<td>com\deitel\advjhtpl\jini\seminar\client\</td>
</tr>
<tr>
<td>UnicastSeminarInfoClient$1.class</td>
<td>com\deitel\advjhtpl\jini\seminar\client\</td>
</tr>
<tr>
<td>UnicastSeminarInfoClient$2.class</td>
<td>com\deitel\advjhtpl\jini\seminar\client\</td>
</tr>
</tbody>
</table>

**Fig. 22.24** SeminarClient.jar contents.

### SeminarServiceDownload.jar contents.

<table>
<thead>
<tr>
<th>Class File</th>
<th>Directory in SeminarServiceDownload.jar</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeminarProxy.class</td>
<td>com\deitel\advjhtpl\jini\seminar\service\</td>
</tr>
<tr>
<td>BackendInterface.class</td>
<td>com\deitel\advjhtpl\jini\seminar\service\</td>
</tr>
<tr>
<td>SeminarInfo_Stub.class</td>
<td>com\deitel\advjhtpl\jini\seminar\service\</td>
</tr>
</tbody>
</table>

**Fig. 22.25** SeminarServiceDownload.jar contents.

**Fig. 22.26** Web server configuration for SeminarInfo service.
Start the SeminarInfo service by launching SeminarInfoService. Be sure that jini-core.jar, jini-ext.jar and sun-util.jar are in your CLASSPATH, and enter the following command at the command prompt:

```
java -classpath %CLASSPATH%;SeminarService.jar
-Djava.security.policy=\policy
-Djava.rmi.server.codebase=http://hostname:9090/
SeminarServiceDownload.jar
com.deitel.advjhtp1.jini.seminar.service.SeminarInfoService
```

where policy is an appropriate security policy and hostname is the hostname on which the Web server for downloading the Jini service proxy is running. Be sure to specify the proper port number for the Web server that serves SeminarServiceDownload.jar (e.g., 9090).

Start the SeminarInfo Jini client by launching UnicastSeminarInfoService. Ensure that jini-core.jar, jini-ext.jar and sun-util.jar are in your CLASSPATH, and that none of the SeminarInfo service class files are in your CLASSPATH. Type the following from a command prompt:

```
java -classpath %CLASSPATH%;SeminarClient.jar
-Djava.security.policy=\policy
UnicastSeminarInfoClient hostname
```

where policy is an appropriate security policy and hostname is the hostname of the computer providing lookup services. Figure 22.27 shows the UnicastSeminarInfoClient application output.

**Common Programming Error 22.4**

Placing the JAR file or classes for the SeminarInfo service in the Jini client’s CLASSPATH prevents the client from downloading the class files at runtime using network class loading.

---

**Fig. 22.27** UnicastSeminarInfoClient application output.
22.8 Introduction to High-Level Helper Utilities

One can build a complete Jini-aware service using previously introduced technology. However, Jini helper utilities simplify the process of developing the Jini applications. These helper utilities provide high-level management capabilities. As we proceed, you will see how these utilities make developing and using Jini services easier.

22.8.1 Discovery Utilities

As you now know, before a client or a server can interact with a lookup service, it must first discover the lookup service. Figure 22.13 introduced a low-level discovery utility, \texttt{LookupDiscovery}. This section discusses two high-level discovery utilities—class \texttt{net.jini.discovery.LookupLocatorDiscovery} and class \texttt{net.jini.discovery.LookupDiscoveryManager}.

\textit{LookupLocatorDiscovery} Utility

In the \textit{UnicastDiscovery} example (Fig. 22.10), we used class \texttt{LookupLocator} to discover lookup services on a known host. Using that technique to discover lookup services located on several known hosts would require several \texttt{LookupLocator}s—one for each host that provides lookup services. Class \texttt{LookupLocatorDiscovery} enables a Jini service or client to discover lookup services on multiple known hosts more easily. Class \texttt{LookupLocatorDiscovery} uses \texttt{DiscoveryEvents} to notify the Jini service or client of discovered lookup services. This is similar to the task class \texttt{LookupDiscovery} performs for multicast discovery.

Class \texttt{UnicastDiscoveryUtility} (Fig. 22.28) uses class \texttt{LookupLocatorDiscovery} to perform unicast lookup-service discovery on multiple known hosts. Class \texttt{UnicastDiscoveryUtility} implements interface \texttt{DiscoveryListener} to receive \texttt{DiscoveryEvents} from the \texttt{LookupLocatorDiscovery} object.

The \texttt{UnicastDiscoveryUtility} constructor (lines 31–62) takes a \texttt{String} array as an argument containing a list of \texttt{jini:} URLs on which to perform unicast discovery. Lines 42–43 create an array of \texttt{LookupLocator}s and lines 49–50 create a \texttt{LookupLocator} object for each URL in array \texttt{urls}. Line 53 registers the \texttt{UnicastDiscoveryUtility} object as a \texttt{DiscoveryListener} for the \texttt{LookupLocatorDiscovery} object.

```java
// UnicastDiscoveryUtility.java
// Demonstrating how to locate multiple lookup services
// using LookupLocatorDiscovery utility
package com.deitel.advjhtp1.jini.utilities.discovery;

// Java core packages
import java.rmi.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

// UnicastDiscoveryUtility uses class LookupLocatorDiscovery to facilitate lookup service discovery (part 1 of 4).
```
public class UnicastDiscoveryUtility extends JFrame
    implements DiscoveryListener {

    private JTextArea outputArea = new JTextArea(10, 20);

    // UnicastDiscoveryUtility constructor
    public UnicastDiscoveryUtility( String urls[] )
    {
        super( "UnicastDiscoveryUtility" );

        getContentPane().add( new JScrollPane( outputArea ),
            BorderLayout.CENTER );

        // discover lookup services using LookupLocatorDiscovery
        try {

            // create LookupLocator for each URL
            LookupLocator locators[] =
                new LookupLocator[ urls.length ];

            for ( int i = 0; i < locators.length ; i++ )
                locators[ i ] = new LookupLocator( urls[ i ] );

            // create LookupLocatorDiscovery object
            LookupLocatorDiscovery locatorDiscovery =
                new LookupLocatorDiscovery( locators );

            // register DiscoveryListener
            locatorDiscovery.addDiscoveryListener( this );

        } // end try

        // handle invalid Jini URL
        catch ( MalformedURLException exception ) {
            exception.printStackTrace();
        }
    } // end UnicastDiscoveryUtility constructor

Fig. 22.28 UnicastDiscoveryUtility uses class LookupLocatorDiscovery to facilitate lookup service discovery (part 2 of 4).
// receive notification of found lookup services
public void discovered( DiscoveryEvent event )
{
    // get the proxy registrars for those services
    ServiceRegistrar[] registrars = event.getRegistrars();

    // display information for each lookup service found
    for ( int i = 0; i < registrars.length ; i++ )
        displayServiceDetails( registrars[ i ] );
}

// display details of given ServiceRegistrar
private void displayServiceDetails( ServiceRegistrar registrar )
{
    try {
        final StringBuffer buffer = new StringBuffer();

        // get the hostname and port number
        buffer.append( "Lookup Service: ");
        buffer.append( "\n Host: " +
                        registrar.getLocator().getHost() );
        buffer.append( "\n Port: " +
                        registrar.getLocator().getPort() );
        buffer.append( "\n Group support: " );

        // get lookup service groups
        String[] groups = registrar.getGroups();

        // get group names; if empty write public
        for ( int i = 0; i < groups.length ; i++ ) {
            if ( groups[ i ].equals( "" ) )
                buffer.append( "public," );
            else
                buffer.append( groups[ i ] + "," );
        }

        buffer.append( "\n\n" );

        // append information to outputArea
        SwingUtilities.invokeLater(
            new Runnable() {
                public void run()
                {
                    outputArea.append( buffer.toString() );
                }
            });
    }
}

Fig. 22.28 UnicastDiscoveryUtility uses class LookupLocatorDiscovery to facilitate lookup service discovery (part 3 of 4).
outputArea.setCaretPosition(outputArea.getText().length());
}
})(); // end call to invokeLater

// handle exception communicating with lookup service
// ignore discarded lookup services
public void discarded(DiscoveryEvent event) {}

// launch UnicastDiscoveryUtility application
public static void main(String args[]) {
    // set SecurityManager
    if (System.getSecurityManager() == null)
        System.setSecurityManager(new RMISecurityManager());
    // check command-line arguments for hostnames
    if (args.length < 1) {
        System.err.println("Usage: java UnicastDiscoveryUtility " +
            "jini://hostname:port [ jini://hostname:port ] ... ");
    }
    // launch UnicastDiscoveryUtility for set of hostnames
    else {
        UnicastDiscoveryUtility unicastUtility =
            new UnicastDiscoveryUtility(args);
        unicastUtility.setDefaultCloseOperation(EXIT_ON_CLOSE);
        unicastUtility.setSize(300,300);
        unicastUtility.setVisible(true);
    }
} // end method main

Fig. 22.28 UnicastDiscoveryUtility uses class LookupLocatorDiscovery to facilitate lookup service discovery (part 4 of 4).

Class LookupLocatorDiscovery invokes method discovered (lines 65–74) when it discovers new lookup services. Line 68 retrieves an array of ServiceRegistrars from the DiscoveryEvent object and line 72 invokes method displayServiceDetails to show information about each discovered lookup service. Method displayServiceDetails (lines 77–129) places host, port and group information...
about a `ServiceRegistrar` in a `StringBuffer` (lines 83–103) and appends the text in that `StringBuffer` to `outputArea` (lines 106–120).

To demonstrate the `UnicastDiscoverUtility` application on a single computer, start multiple instances of the Jini lookup service. Remember that you must specify a different log directory for each Reggie instance. After starting several Reggie instances, run the `MulticastDiscovery` application of Fig. 22.13 to obtain the various port numbers on which Reggies are running. Recall that `UnicastDiscoverUtility` uses unicast discovery, therefore you must specify the hostnames and port numbers for the computers running the lookup services. Using the `MulticastDiscovery` application is an easy way to determine these hostnames and port numbers for testing class `UnicastDiscoveryUtility`. Figure 22.29 shows sample output from application `MulticastDiscovery` with four lookup services running on the local machine. Note that each lookup service uses a different port number. We use these port numbers when specifying `jini` URLs for application `UnicastDiscoveryUtility`.

After running the `MulticastDiscovery` application to find the available lookup services, enter the following to launch `UnicastDiscoveryUtility`:

```
java -Djava.security.policy=policy
   com.deitel.advjhtp1.jini.utilities.discovery.
   UnicastDiscoveryUtility
       jini://hostname:4160
```

where `policy` is an appropriate security policy and `hostname` is the hostname running the lookup services. Remember to replace the port numbers in the preceding command with the port numbers that application `MulticastDiscovery` found on your computer. Figure 22.30 shows the `UnicastDiscoveryUtility` application output.
**LookupDiscoveryManager Utility**

Class `LookupDiscoveryManager` provides flexible lookup service discovery by enabling Jini applications and clients to perform both unicast and multicast lookup service discovery using a single class. Class `LookupDiscoveryManager` combines the functionality provided in classes `LookupLocatorDiscovery` (for unicast discovery) and `LookupDiscovery` (for multicast discovery).

Class `GeneralDiscoveryUtility` (Fig. 22.31) performs unicast and multicast discovery using class `LookupDiscoveryManager`. Lines 38–51 create and lay out two `JTextArea`s—one for displaying unicast discovery notifications and one for displaying multicast discovery notifications. Lines 55–60 create an array of `LookupLocator`s and populate the array with `jini:` URLs from the command-line arguments. Lines 63–64 create a new `LookupDiscoveryManager`. The `LookupDiscoveryManager` constructor takes as its first argument an array of group names for performing multicast discovery. The constant `DiscoveryGroupManagement.ALL_GROUPS` specifies that the `LookupDiscoveryManager` should discover all lookup services. This constant is equivalent to passing `null` as the first argument. If the first argument is an empty `String` array, no multicast discovery is performed. The second argument to the `LookupDiscoveryManager` constructor is an array of `LookupLocator`s for performing unicast discovery. If the `LookupLocator` array is `null` or empty, no unicast discovery is performed. The final argument is the `DiscoveryListener` to which discovery notifications should be sent.

Method `discovered` (lines 80–101) receives discovery notifications from the `LookupDiscoveryManager`. Line 83 retrieves an array of discovered `ServiceRegistrar`s from the `DiscoveryEvent` object. Line 89 invokes method `getFrom` of class `LookupDiscoveryManager` to determine which form of discovery—unicast or multicast—discovered the given `ServiceRegistrar`. Constant `LookupDiscoveryManager.FROM_GROUP` (line 90) identifies `ServiceRegistrar`s discovered through multicast, or group, discovery. If the `ServiceRegistrar` was discovered through multicast discovery, lines 92–93 invoke method `displayServiceDetails` to...
display information about the `ServiceRegistrar` in `multicastArea`. If the `ServiceRegistrar` was not discovered through multicast discovery, line 98 invokes method `displayServiceDetails` to display the `ServiceRegistrar` information in `unicastArea`.

```java
// GeneralDiscoveryUtility.java
// GeneralDiscoveryUtility demonstrates using class
// LookupDiscoveryManager for performing multicast
// and unicast discovery.
package com.deitel.advjhtp1.jini.utilities.discovery;

// Java core packages
import java.rmi.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

// Java standard extensions
import javax.swing.*;
import javax.swing.border.*;

// Jini core packages
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;

// Jini extension packages
import net.jini.discovery.*;

public class GeneralDiscoveryUtility extends JFrame implements DiscoveryListener {

    private LookupDiscoveryManager lookupManager;
    private JTextArea multicastArea = new JTextArea( 15, 20);
    private JTextArea unicastArea = new JTextArea( 15, 20);

    // GeneralDiscoveryUtility constructor
    public GeneralDiscoveryUtility( String urls[] )
    {
        super( "GeneralDiscoveryUtility" );

        // lay out JTextAreas
        JPanel multicastPanel = new JPanel();
        multicastPanel.setBorder( new TitledBorder( "Multicast (Group) Notifications" ) );
        multicastPanel.add( new JScrollPane( multicastArea ) );

        JPanel unicastPanel = new JPanel();
        unicastPanel.setBorder( new TitledBorder( "Unicast (Locator) Notifications" ) );

        // GeneralDiscoveryUtility uses class LookupDiscoveryManager to perform both unicast and multicast lookup service discovery (part 1 of 4).
unicastPanel.add( new JScrollPane( unicastArea ) );

Container contentPane = getContentPane();
contentPane.setLayout( new FlowLayout() );
contentPane.add( unicastPanel );
contentPane.add( multicastPanel );

// get LookupLocators and LookupDiscoveryManager
try {
    LookupLocator locators[] =
        new LookupLocator[ urls.length ];

    // get array of LookupLocators
    for ( int i = 0; i < urls.length ; i++ )
        locators[ i ] = new LookupLocator( urls[ i ] );

    // instantiate a LookupDiscoveryManager object
    lookupManager = new LookupDiscoveryManager(
        DiscoveryGroupManagement.ALL_GROUPS, locators, this );
} // end GeneralDiscoveryUtility constructor

// receive notifications of discovered lookup services.
public void discovered( DiscoveryEvent event ) {

    // get the proxy registrars for those services
    ServiceRegistrar[] registrars = event.getRegistrars();

    // display information for each lookup service found
    for ( int i = 0; i < registrars.length ; i++ ) {

        // display multicast results in multicastArea
        if ( lookupManager.getFrom( registrars[ i ] ) ==
            LookupDiscoveryManager.FROM_GROUP ) {

            displayServiceDetails( registrars[ i ],
                multicastArea );
        }
    }

Fig. 22.31 GeneralDiscoveryUtility uses class LookupDiscoveryManager to perform both unicast and multicast lookup service discovery (part 2 of 4).
// display unicast results in unicastArea
else
displayServiceDetails( registrars[ i ], unicastArea );
}
} // end method discovered

// display details of given ServiceRegistrar
private void displayServiceDetails(
    ServiceRegistrar registrar, final JTextArea outputArea )
{
    try {
        final StringBuffer buffer = new StringBuffer();

        // get hostname and port number
        buffer.append( "Lookup Service: " );
        buffer.append( "\n Host: " +
            registrar.getLocator().getHost() );
        buffer.append( "\n Port: " +
            registrar.getLocator().getPort() );
        buffer.append( "\n Group support: " );

        // get lookup service groups
        String[] groups = registrar.getGroups();

        // get group names; if empty write public
        for ( int i = 0; i < groups.length ; i++ ) {

            if ( groups[ i ].equals( "" ) )
                buffer.append( "public," );
            else
                buffer.append( groups[ i ] + "," );
        }

        buffer.append( "\n\n" );

        // append information to outputArea
        SwingUtilities.invokeLater(
            new Runnable() {

                // append text and update caret position
                public void run()
                {
                    outputArea.append( buffer.toString() );
                    outputArea.setText( outputArea.getText().length() );
                }
            });
    }
}

Fig. 22.31 GeneralDiscoveryUtility uses class LookupDiscoveryManager to perform both unicast and multicast lookup service discovery (part 3 of 4).
Method `displayServiceDetails` (lines 104–159) takes as arguments a `ServiceRegistrar` and a `JTextArea` in which to display that `ServiceRegistrar`’s information. Lines 111–131 append the `ServiceRegistrar`’s information to a `StringBuffer`. Lines 134–148 invoke the `invokeLater` method of class `SwingUtilities` to append the `ServiceRegistrar` information to the appropriate `JTextArea`.

Method `main` (lines 163–176) installs an `RMISecurityManager` (line 167) and creates a new `GeneralDiscoveryUtility` instance, passing the array of command-line arguments to the constructor (lines 170–171). If the user does not supply any `jini:` URLs as command-line arguments, `GeneralDiscoveryUtility` will not perform unicast discovery. To execute `GeneralDiscoveryUtility`, enter the following at a command prompt:

```
java -Djava.security.policy=.policy
com.deitel.advjhtp1.jini.utilities.discovery.
GeneralDiscoveryUtility jini://hostname:4160
```
where *policy* is an appropriate security policy, *hostname* is the hostname for a known computer running a lookup service and **4160** is the default lookup service port number. Figure 22.32 shows the output from *GeneralDiscoveryUtility* with several lookup services running on the local machine.

### 22.8.2 Entry Utilities

**Entry** attributes specify characteristics of Jini services. We used the **Name Entry** in Fig. 22.21 to provide a name for the *SeminarInfo* Jini service. **Entry** attributes help clients identify Jini services. By attaching attributes to services, service providers can publish services with detailed information, such as the service location and the functionality of the service. Jini provides seven common attributes (Fig. 22.33).

#### Table: Standard Jini Entry Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Specifies the physical location of a service. (e.g., the city and street address of an Automated Teller Machine).</td>
</tr>
<tr>
<td>Comment</td>
<td>Provides general descriptive comments about a service.</td>
</tr>
<tr>
<td>Location</td>
<td>Provides more detailed location information, such as floor, suite and room number.</td>
</tr>
<tr>
<td>Name</td>
<td>Provides the service’s name, suitable for use by people to identify a service (e.g., “BankABC ATM”).</td>
</tr>
<tr>
<td>ServiceInfo</td>
<td>Provides basic information about a service. For example, the manufacturer and model number of a printer.</td>
</tr>
<tr>
<td>ServiceType</td>
<td>Provides a human-understandable description of the service type (e.g., “Print Queue”).</td>
</tr>
<tr>
<td>Status</td>
<td>Describes the current status of the service in varying levels of severity.</td>
</tr>
</tbody>
</table>

![GeneralDiscoveryUtility application output](image)
Developers also can create custom attributes for Jini services. Class `SeminarProvider` (Fig. 22.34) is an `Entry` attribute that gives the name of the company that provides seminars for a given `SeminarInfo` service. Class `SeminarProvider` extends class `AbstractEntry` and implements interface `ServiceControlled`. `AbstractEntry` is a basic implementation of interface `Entry`. By implementing interface `ServiceControlled`, class `SeminarProvider` indicates that the service itself controls the `SeminarProvider` attribute.

An `Entry` class must supply a no-argument constructor (line 12). Also, instance variables must be `public` references to `Serializable` objects, so clients can perform searches using these instance variables. Line 12 declares `public String` reference `providerName`, which contains the name of the organization providing the seminars.

To use the `SeminarProvider` attribute for our `SeminarInfo` Jini service, we need to modify class `SeminarInfoService`, which registers the `SeminarInfo` service with `ServiceRegistrar`es. Replace line 43 of Fig. 22.21 with the line

```java
entries[ 0 ] = new com.deitel.advjhtp1.jini.utilities.entry.SeminarProvider(
    "Deitel" );
```

We also must modify `UnicastSeminarInfoClient` to search for the `SeminarInfoServices` using the new `SeminarProvider` attribute. Replace line 110 of Fig. 22.22 with the line

```java
Entry[] attribute = new Entry[] { new com.deitel.advjhtp1.jini.utilities.entry.SeminarProvider(
    "Deitel" ) ;
```

Fig. 22.34 SeminarProvider subclass of `Entry` for describing the Seminar provider as a Jini attribute.
Compile and run the Jini service and `UnicastSeminarInfoClient` to locate the `SeminarInfo` service using the new `SeminarProvider` attribute. Be sure to include `SeminarProvider.class` in the service and client JAR files.

### 22.8.3 Lease Utilities

Jini uses *leasing* to ensure integrity in distributed systems built with Jini. Recall that Jini services register with lookup services to make the Jini service’s functionality available to other members in the Jini community. If all goes well, other members of the Jini community use the service and the service stays up and running perpetually. However, in reality, services fail for many reasons. Network outages can make a service unreachable. A physical device associated with a service (e.g., a printer) might need repairs. The service itself could encounter an unrecoverable exception. In these and many other situations, a service could become unavailable and that service might not be able to unregister itself from lookup services to prevent other clients from attempting to use that service.

One goal of Jini technology is to make Jini communities “self-healing” and able to recover from common problems, such as network outages, hardware failures and software failures. Therefore, when a Jini service registers with a lookup service, the registration is not permanent. The registration is leased for a specific amount of time, after which the lookup service revokes the registration. This prevents problematic services from disrupting the entire Jini community. If a Jini service fails, the Jini service’s lease eventually expires, and lookup services will no longer provide the failed Jini service to clients.

The leasing strategy that Jini employs is strict—if a Jini service does not renew its lease, the lookup service terminates the registration when the lease expires and the service becomes unavailable to clients. Therefore, developers must ensure that their Jini services manage registration leases to ensure that services do not lose their registrations prematurely.

Our `SeminarInfo` Jini service does not perform any lease maintenance. After the `SeminarInfo` service’s first lease expires (i.e., 10 minutes), the `SeminarInfo` service is no longer available to clients. The service itself continues executing, but the lookup services with which the service registered will have terminated the registrations.

Class `LeaseRenewalManager` is a Jini utility class that enables services to manage their leases to ensure that the service’s leases do not expire prematurely. Class `SeminarInfoLeaseService` (Fig. 22.35) uses class `LeaseRenewalManager` to manage leases for the `SeminarInfo` service. Class `SeminarInfoLeaseService` is similar to class `SeminarInfoService` (Fig. 22.21), so we concentrate only on lease management in this example.

```java
// SeminarInfoLeaseService.java

// SeminarInfoLeaseService discovers lookup services, registers
// the SeminarInfo service, and creates a LeaseRenewalManager
// to maintain the SeminarInfo service lease.
package com.deitel.advjhtp1.jini.utilities.leasing;

// Java core packages
import java.rmi.RMISecurityManager;
```

Fig. 22.35 `SeminarInfoLeaseService` uses class `LeaseRenewalManager` to manage `SeminarInfo` service leasing (part 1 of 4).
import java.rmi.RemoteException;
import java.io.IOException;

// Jini core packages
import net.jini.core.lookup.*;
import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;

// Jini extension packages
import net.jini.discovery.*;
import net.jini.lookup.entry.Name;
import net.jini.lease.LeaseRenewalManager;

// Deitel packages
import com.deitel.advjhtp1.jini.seminar.service.*;
import com.deitel.advjhtp1.jini.utilities.entry.SeminarProvider;

public class SeminarInfoLeaseService implements DiscoveryListener {

    private LookupDiscovery discover;
    private ServiceItem item;
    private static final int LEASETIME = 10 * 60 * 1000;

    // SeminarInfoLeaseService constructor
    public SeminarInfoLeaseService()
    {
        // search for lookup services with public group
        try {
            discover = new LookupDiscovery( new String[] { "" } );
            // register DiscoveryListener
            discover.addDiscoveryListener( this );
        }
        catch ( IOException exception ) {
            exception.printStackTrace();
        }

        // handle exception creating LookupDiscovery
        catch ( IOException exception ) {
            exception.printStackTrace();
        }

        // create and set Entry name for service
        Entry[] entries = new Entry[ 1 ];
        entries[ 0 ] = new SeminarProvider( "Deitel" );

        // specify the service's proxy and entry
        item = new ServiceItem( null, createProxy(), entries );
    }

    // receive notifications of discovered lookup services
    public void discovered ( DiscoveryEvent event )
    {
        ServiceRegistrar[] registrars = event.getRegistrars();
    }

Fig. 22.35 SeminarInfoLeaseService uses class LeaseRenewalManager to manage SeminarInfo service leasing (part 2 of 4).
// register the service with the lookup service
for ( int i = 0; i < registrars.length; i++ ) {
    ServiceRegistrar registrar = registrars[ i ];

    // register the service with the lookup service
    try {
        ServiceRegistration registration =
            registrar.register( item, LEASETIME );

        // create LeaseRenewalManager
        LeaseRenewalManager leaseManager =
            new LeaseRenewalManager();

        // renew SeminarInfo lease indefinitely
        leaseManager.renewUntil( registration.getLease(),
            Lease.FOREVER, null );
    }  // end try

    // handle exception registering ServiceItem
    catch ( RemoteException exception ) {
        exception.printStackTrace();
    }
}  // end for

} // end method discovered

// ignore discarded lookup services
public void discarded( DiscoveryEvent event ) {}

// create seminar service proxy
private SeminarInterface createProxy()
{
    // get BackendInterface reference to SeminarInfo
    try {
        BackendInterface backInterface = new SeminarInfo();

        return new SeminarProxy( backInterface );
    }

    // handle exception creating SeminarProxy
    catch ( RemoteException exception ) {
        exception.printStackTrace();
    }

    return null;
}

} // end method createProxy

Fig. 22.35 SeminarInfoLeaseService uses class LeaseRenewalManager to manage SeminarInfo service leasing (part 3 of 4).
In method `discovered`, lines 69–70 register the `SeminarInfo` service’s `ServiceItem` with the discovered `ServiceRegistrar`. Lines 73–74 create a new `LeaseRenewalManager`, which class `SeminarInfoLeaseService` uses to manage the `SeminarInfo` service’s lease. Lines 77–78 invoke method `renewUntil` of class `LeaseRenewalManager`. Method `renewUntil` takes as its first argument the `Lease` object to be renewed. In this example, we obtain the `Lease` object by invoking method `getLease` of class `ServiceRegistration`. The second argument specifies the desired expiration time (in milliseconds) for renewed `Leases`. Line 78 specifies the constant `Lease.FOREVER` to request a `Lease` that never expires. This does not guarantee that the lookup service will provide an everlasting `Lease`—the lookup service is free to grant a lease that is shorter than the requested length. The third argument to method `renewUntil` is the `LeaseListener`, to notify of problems encountered when renewing the `Lease`. We pass `null` as the third argument to disregard such notifications.

To run the `SeminarInfo` Jini service with lease management, create a new JAR file named `SeminarServiceWithLeasing.jar`. Figure 22.36 shows the contents of `SeminarServiceWithLeasing.jar`. Note that this JAR file replaces `SeminarInfoService.class` with `SeminarInfoLeaseService.class`. Note also that
this JAR file includes `SeminarProvider.class`, which is our custom Entry object for the SeminarInfo service.

After packaging the classes in `SeminarServiceWithLeasing.jar`, run the new version of the service by typing the following command at a command prompt:

```
java -classpath %CLASSPATH%;SeminarServiceWithLeasing.jar
   -Djava.security.policy=policy
   -Djava.rmi.server.codebase=http://hostname:9090/
   SeminarServiceDownload.jar
   com.deitel.advjhtp1.jini.utilities.leasing.
   SeminarInfoLeaseService
```

where `policy` is an appropriate security policy and `hostname` is the hostname where the Web server for downloading the SeminarInfo service proxy is running. The LeaseRenewalManager will renew the SeminarInfo service’s lease to maintain the service’s lookup service registrations.

### 22.8.4 JoinManager Utility

As we have seen, making a Jini service available in a Jini community requires several steps. The service must discover lookup services, register with discovered lookup services and maintain registration leases. Class JoinManager is a utility class that facilitates the process of deploying a Jini service by performing lookup discovery, service registration and lease management in a single class.

<table>
<thead>
<tr>
<th>Class File</th>
<th>Directory in SeminarServiceWithLeasing.jar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seminar.class</td>
<td>com\deitel\advjhtp1\jni\seminar\</td>
</tr>
<tr>
<td>SeminarInterface.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service|</td>
</tr>
<tr>
<td>SeminarProxy.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service|</td>
</tr>
<tr>
<td>BackendInterface.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service|</td>
</tr>
<tr>
<td>SeminarInfo.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service|</td>
</tr>
<tr>
<td>SeminarInfo_Stub.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service|</td>
</tr>
<tr>
<td>SeminarProvider.class</td>
<td>com\deitel\advjhtp1\jni\utilities\entry|</td>
</tr>
<tr>
<td>SeminarInfoLeaseService.class</td>
<td>com\deitel\advjhtp1\jni\utilities\leasing|</td>
</tr>
</tbody>
</table>

*Fig. 22.36 SeminarServiceWithLeasing.jar contents.*
Class SeminarInfoJoinService (Fig. 22.37) uses class JoinManager to deploy the SeminarInfo service. Lines 38–40 create a LookupDiscoveryManager that the JoinManager will use to discover lookup services. We pass as the first argument to the LookupDiscoveryManager constructor a String array with a single element, which is an empty String. This argument specifies that the LookupDiscoveryManager should perform multicast discovery for lookup services that support the “public” group. For the second and third arguments, line 40 passes the value null. These arguments disable unicast discovery and specify a null DiscoveryListener, respectively. The JoinManager handles the discovery process, so class SeminarInfoJoinService need not handle DiscoveryEvents.

Lines 43–44 create a new Entry array with a single SeminarProvider element that specifies the provider of seminars for the SeminarInfo service. Lines 47–49 create a new instance of class JoinManager to discover lookup services, register the service and maintain the service’s registration leases. Line 47 invokes method createProxy to create a SeminarProxy for the SeminarInfo service. The second argument to the JoinManager constructor is the array of Entry attributes that describe the service. The third argument is a reference to a ServiceIDListener. When the JoinManager registers the Jini service with a lookup service, the JoinManager notifies the ServiceIDListener of the service ID that the lookup service assigns to the Jini service. The fourth argument is a DiscoveryManagement object for discovering lookup services. For this example, we pass the LookupDiscoveryManager created on lines 38–40. The final argument to the JoinManager constructor is a LeaseRenewalManager for maintaining the service’s registration leases.

Fig. 22.37 SeminarInfoJoinService uses class JoinManager to facilitate registering the SeminarInfo service and manage its leasing (part 1 of 3).
// Deitel packages
import com.deitel.advjhtp1.jini.seminar.service.*;
import com.deitel.advjhtp1.jini.utilities.entry.*;

public class SeminarInfoJoinService implements ServiceIDLListener {

    // SeminarInfoJoinService constructor
    public SeminarInfoJoinService()
    {
        // use JoinManager to register SeminarInfo service
        // and manage lease
        try {
            // create LookupDiscoveryManager for discovering
            // lookup services
            LookupDiscoveryManager lookupManager =
                new LookupDiscoveryManager( new String[] { "" },
                null, null );

            // create and set Entry name for service
            Entry[] entries = new Entry[ 1 ];
            entries[ 0 ] = new SeminarProvider( "Deitel" );

            // create JoinManager
            JoinManager manager = new JoinManager( createProxy(),
                entries, this, lookupManager,
                new LeaseRenewalManager() );
        }

        // handle exception creating JoinManager
        catch ( IOException exception ) {
            exception.printStackTrace();
        }
    }

    // create seminar service proxy
    private SeminarInterface createProxy()
    {
        // get SeminarProxy for SeminarInfo service
        try {
            return new SeminarProxy( new SeminarInfo() );
        }

        // handle exception creating SeminarProxy
        catch ( RemoteException exception ) {
            exception.printStackTrace();
        }

        return null;
    }

} // end SeminarInfoJoinService constructor

// Fig. 22.37 SeminarInfoJoinService uses class JoinManager to facilitate
registering the SeminarInfo service and manage its leasing (part 2 of 3).
Method `serviceIDNotify` (lines 77–80) is required by interface `ServiceIDListener`. The `JoinManager` invokes method `serviceIDNotify` to notify a `ServiceIDListener` that a lookup service has assigned a service ID to the Jini service. Line 79 simply prints out the service ID.

Method `main` (lines 83–92) sets the `RMISecurityManager` and launches the `SeminarInfoJoinService` application. Note that method `main` does not use a `keepAliveObject` to keep the application running, as was required in previous examples. The `JoinManager` keeps the application alive.

To run the `SeminarInfo` service using the `JoinManager`, create the JAR file `SeminarServiceJoinManager.jar` with the contents listed in Figure 22.38.

### Fig. 22.37 SeminarInfoJoinService uses class JoinManager to facilitate registering the SeminarInfo service and manage its leasing (part 3 of 3).

```java
// receive notification of ServiceID assignment
public void serviceIDNotify( ServiceID serviceID )
{
    System.err.println( "Service ID: " + serviceID );
}

// launch SeminarInfoJoinService
public static void main( String args[] )
{
    // set SecurityManager
    if ( System.getSecurityManager() == null ) {
        System.setSecurityManager( new RMISecurityManager() );
    }

    // create SeminarInfoJoinService
    new SeminarInfoJoinService();
}
```

Service ID: 0084d3a0-4bbe-4b76-aa0b-f73294738fec

### Class File Directory in SeminarServiceJoinManager.jar

<table>
<thead>
<tr>
<th>Class File</th>
<th>Directory in SeminarServiceJoinManager.jar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seminar.class</td>
<td>com\deitel\advjhtp1\jni\seminar\</td>
</tr>
<tr>
<td>SeminarInterface.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service\</td>
</tr>
<tr>
<td>SeminarProxy.class</td>
<td>com\deitel\advjhtp1\jni\seminar\service\</td>
</tr>
</tbody>
</table>

### Fig. 22.38 SeminarServiceJoinManager.jar contents (part 1 of 2).
After packaging the classes in SeminarServiceJoinManager.jar, run the new version of the service by typing the following at a command prompt:

```
java -classpath %CLASSPATH%;SeminarServiceJoinManager.jar
-Djava.security.policy=policy
-Djava.rmi.server.codebase=http://hostname:9090/
SeminarServiceDownload.jar
SeminarInfoJoinService
```

where policy is an appropriate security policy and hostname is the hostname where the Web server for downloading the SeminarInfo service proxy is running. Figure 22.37 shows a sample service ID output from application SeminarInfoJoinService.

### 22.8.5 Service Discovery Utilities

Complex Jini clients often have specific requirements for the Jini services they employ. To satisfy these requirements, the Jini client often must work with sets of Jini services. The client searches through these services to locate the particular service that can satisfy the client’s needs. For example, a Jini client that provides users with information about printers available in a given office needs a set of services for the available printers. This printer-monitoring program would need to be aware of the status of each printer to know when a new printer has been added. The service also should be able to search among the printers for particular features (color support, print quality, capacity, speed etc.).

Class net.jini.lookup.ServiceDiscoveryManager provides Jini clients with a richer set of service and lookup-service management features than interface ServiceRegistrar provides. Class ServiceDiscoveryManager facilitates discovering available services and enables clients to perform finer-grained searches than are possible with the ServiceRegistrar interface. Jini clients also can use class ServiceDiscoveryManager to enhance application performance by maintaining a local cache of services. There are three primary ways in which Jini clients use class ServiceDiscoveryManager—creating a local cache of services, receiving event notifications
when services become available or unavailable and performing detailed searches not possible with simple ServiceTemplates.

Class ServiceDiscoveryManager can enhance Jini-client performance by creating a local cache of discovered services. This local cache—implemented as a net.jini.lookup.LookupCache—enables the client to perform additional service lookups without incurring the network overhead of a remote call to a lookup service. When the client needs a particular service that is in the LookupCache, the client simply invokes method lookup of interface LookupCache to retrieve the service from the local cache.

Jini clients also can use a LookupCache retrieved from a ServiceDiscoveryManager to receive notifications related to a set of services. By implementing interface ServiceDiscoveryListener and registering with a LookupCache, a Jini client can receive events indicating when a particular service has been discovered, when a service’s attributes have changed and when the service is removed from the LookupCache. This event notification is particularly useful for Jini clients that monitor available resources, such as our printer-monitoring example.

Class ServiceDiscoveryManager also provides an enhanced interface that enables Jini clients to search for services using more specific search criteria. Jini clients can use class ServiceDiscoverManager with implementations of interface ServiceItemFilter to locate services whose attribute values fall within a particular range. For example, a Jini client could use a ServiceItemFilter to locate all automated teller machines in the area whose service charge is less than two dollars. Such a specific query is not possible using the standard ServiceTemplate matching available through interface ServiceRegistrar.

For more information on class ServiceDiscoveryManager, please see the Jini API documentation included with the Jini Technology Core Platform.

### 22.9 Internet and World Wide Web Resources

- **www.jini.org**
  Home of the Jini community.

- **www.sun.com/jini/specs/jini1.1html/coreTOC.html**
  The site for Jini Technology Core Platform Specification.

- **www.sun.com/jini/specs/jini1.1html/collectionTOC.html**
  This site contains a collection of Jini Technology Helper Utilities and Services Specifications.

- **www.sun.com/jini/specs/jini1.1html/jsTOC.html**
  This site provides JavaSpaces Service Specification.

- **developer.java.sun.com/developer/products/jini/installation.index.html**
  This site provides installation instructions for Jini technology.

### SUMMARY

- Many network devices provide services to network clients.
- Each service has a well-defined interface.
- To use a service, a client must be able to discover that a service exists and must know the interface for interacting with the service.
- Jini extends RMI to provide services to a network.
• Jini services are plug-and-play—clients can discover services on the network dynamically, transparently download classes required to use those services, then begin interacting with those services.

• RMI’s dynamic class-downloading capability enables Jini clients to use services without installing special driver software for those services in advance.

• For Jini clients to discover and use Jini services, standardized interfaces for common services must be developed.

• The basic software requirements for Jini include the Java 2 Standard Edition (J2SE) and the Jini Technology Starter Kit. If you are going to write commercial Jini services and want to test their compatibility with the Jini platform, you also need to download the Jini Technology Core Platform Compatibility Kit (Jini TCK).

• The Jini Starter Kit has three components—the Jini Technology Core Platform (JCP), the Jini Technology Extended Platform (JXP) and the Jini Software Kit (JSK). The JCP contains the fundamental Jini interfaces and classes. The JXP provides helper utilities for implementing Jini services and clients. The JSK contains an implementation of the services specified in the JCP and the JXP.

• To compile and execute Jini services and clients the JAR files jini-core.jar, jini-ext.jar and sun-util.jar must be included in the CLASSPATH environment variable. These three JAR files are in the lib directory of the Jini Starter Kit—they correspond to the Jini Technology Core Platform, the Jini Technology Extended Platform and the Jini Software Kit, respectively.

• The Jini distribution comes with three services that must be running correctly before executing Jini applications—a Web server to enable Jini clients to download class files through RMI, so the clients can access Jini services dynamically; the RMI activation daemon (rmid) to enable the RMI infrastructure that allows Jini clients to communicate with Jini services; and a lookup service to maintain information about available Jini services, and to enable clients to discover and use those services. The Web server and rmid must be executing (they can be started in any order) before starting the lookup service.

• The Jini Technology Core Platform implementation includes the StartService GUI tool for launching required services.

• The Jini lookup service is the heart of a Jini community. The process of finding the lookup services and obtaining references to them is called discovery.

• Discovery distinguishes Jini technology from RMI. In RMI, you must know in advance where to register an object. In Jini, you do not need to know where—just how. The discovery process determines where, but hides the details from the developer.

• Discovery can be accomplished using either unicast discovery or multicast discovery.

• Unicast discovery, or locator discovery, enables a Jini service or client to discover lookup services on a specific host.

• Method getRegistrar of class LookupLocator performs unicast discovery. The method returns a ServiceRegistrar, which represents a lookup service. An overloaded version of method getRegistrar takes as an integer argument the maximum number of milliseconds to wait for the unicast discovery to locate a ServiceRegistrar before issuing a timeout.

• Methods getHost and getPort of class LookupLocator retrieve the hostname and port number where a lookup service was discovered.

• Multicast discovery, or group discovery, enables a Jini service or client to discover lookup services when the particular host running the lookup service is not known. A multicast discovery request uses network multicast to discover nearby lookup services. Lookup services periodically issue multicast announcements to notify interested Jini services and clients that the lookup services are available.

• Class net.jini.discovery.LookupDiscovery performs multicast discovery.

• Implementing interface DiscoveryListener enables an object of a class to receive DiscoveryEvents—notifications of discovered lookup services.
• Class **LookupDiscovery** invokes method **discovered** when **LookupDiscovery** locates new lookup services.

• Method **getRegistrars** of **DiscoveryEvent** obtains an array of **ServiceRegistrars**.

• Class **LookupDiscovery** invokes method **discarded** when a lookup service should be discarded because it is no longer available or because it no longer matches the set of groups in which the Jini service or client is interested.

• A Jini service consists of several components, each of which contributes to the flexibility and portability of the Jini architecture. A service proxy is an intermediary between a Jini service and its clients. The service proxy communicates with the actual service implementation through the service’s back-end interface, which defines methods in the service implementation. A separate application discovers lookup services and registers the Jini service, making the service available to Jini clients.

• A Jini client uses the lookup service discovery techniques to discover lookup services. The Jini client then uses the discovered lookup services to locate the desired Jini service. When the lookup service locates the service requested by the Jini client, the lookup service serializes the service proxy and delivers the proxy to the Jini client. The client can then invoke methods defined in the service’s **public** interface directly on the service proxy, which implements that interface. The service proxy communicates with the service implementation through the back-end interface.

• An **Entry** (package **net.jini.core.entry**) describes a service, which enables Jini clients to search for services of a particular description.

• The lookup service requires a **ServiceItem** (package **net.jini.core.lookup**) to register a Jini service.

• Jini helper utilities simplify the process of developing the Jini applications. These helper utilities provide high-level management capabilities.

• Class **LookupLocatorDiscovery** enables a Jini service or client to discover lookup services on multiple known hosts. Class **LookupLocatorDiscovery** uses **DiscoveryEvent**s to notify the Jini service or client of discovered lookup services.

• Class **LookupDiscoveryManager** provides flexible lookup service discovery by enabling Jini applications and clients to perform both unicast and multicast lookup service discovery using a single class.

• **Entry** attributes specify characteristics of Jini services. By attaching attributes to services, service providers can publish services with detailed information, such as the service location and the functionality of the service. Developers also can create custom attributes for Jini services. Class **AbstractEntry** provides a basic implementation of interface **Entry**.

• An **Entry** class must supply a no-argument constructor. Also, instance variables must be **public** references to **Serializable** objects.

• One goal of Jini technology is to make Jini communities “self-healing” and able to recover from common problems, such as network outages, hardware failures and software failures. Therefore, when a Jini service registers with a lookup service, the registration is not permanent. The registration is leased for a specific amount of time, after which the lookup service revokes the registration. This prevents problematic services from disrupting the entire Jini community.

• The leasing strategy that Jini employs is strict—if a Jini service does not renew its lease, the lookup service terminates the registration when the lease expires, making the service unavailable to clients.

• Class **LeaseRenewalManager** is a Jini utility class that enables services to manage their leases to ensure that the service’s leases do not expire prematurely.

• Class **JoinManager** is a utility class that facilitates the process of deploying a Jini service by performing lookup discovery, service registration and lease management in a single class.

• Complex Jini clients often have specific requirements for the Jini services they employ. To satisfy these requirements, the Jini client often must work with sets of Jini services. The client searches...
through these services to locate the particular service that can satisfy the client’s needs. Class `ServiceDiscoveryManager` facilitates discovering available services and enables clients to perform finer-grained searches than are possible with the `ServiceRegistrar` interface.

- There are three primary ways in which Jini clients use class `ServiceDiscoveryManager`—creating a local cache of services, receiving event notifications when services become available or unavailable and performing detailed searches not possible with simple `ServiceTemplates`.
- Class `ServiceDiscoveryManager` can enhance Jini-client performance by creating a local cache of discovered services. This local cache—implemented as a `LookupCache`.

**TERMINOLOGY**

- `serviceAdded` method of `ServiceDiscoveryListener`
- `AbstractEntry` class
- `createLookupCache` method of `ServiceDiscoveryManager`
- `DiscoveryEvent` class
- `DiscoveryListener` interface
- `DiscoveryManagement` class
- `Entry` interface
- `getFrom` method of `LookupDiscoveryManager`
- `getGroups` method of `LookupDiscovery`
- `getHost` method of class `LookupLocator`
- `getPort` method of class `LookupLocator`
- `getRegistrar` method of class `LookupLocator`
- `getRegistrars` method of class `LookupDiscovery`

**SELF-REVIEW EXERCISES**

22.1 Fill in the blanks in each of the following statements:

a) Name three required services for running Jini services and clients: __________, __________, and __________.

b) Two ways to discover lookup services are __________ and __________.
c) To generate the stub file for a remote object, use ________.
d) A service proxy that is exported to the remote client must implement interface ________.
e) Service providers use ________ to describe a service. Jini clients use ________ to find a matching service.

22.2 State whether each of the following is true or false. If false, explain why.
a) Unicast discovery is also known as locator discovery.
b) The JoinManager can discover lookup services, register a service and renew a service’s lease.
c) Class LookupDiscoveryManager can perform only unicast discovery.
d) Jini requires only the RMI activation daemon (rmid) and a Web server.
e) Jini clients must have all the .class files for a Jini service in the local CLASSPATH.

ANSWERS TO SELF-REVIEW EXERCISES

22.1 a) the HTTP Web server, the rmi activation daemon, the lookup service. b) unicast discovery, multicast discovery. c) rmic. d) Serializable. e) ServiceItem, ServiceTemplate.

22.2 a) True. b) True. c) False. Class LookupDiscoveryManager performs both unicast and multicast discovery. d) False. Jini also requires lookup services to enable clients to locate Jini services. e) False. Jini clients require that only the public interface and supporting classes be in the local CLASSPATH. Having the Jini service’s .class files in the client’s CLASSPATH prevents network class loading.

EXERCISES

22.3 Modify class MulticastDiscovery (Fig. 22.13) to perform multicast discovery for lookup services that support any group, not just the public group. Can you use null for a wildcard match?

22.4 Create an application to find all services that are registered with local lookup services.

22.5 Write a currency exchange service using Jini technology. This currency exchange service simply does one function: It exchanges the currency of one country to the currency of another country. The exchange rate can be dynamically loaded from an on-line resource or can be just statically loaded from a file. Create the public interface, service proxy, back-end interface and service implementation.

22.6 Register the currency exchange service with the lookup services on the local machine using JoinManager.

22.7 Create a Jini client that allows a user to use the exchange service. Search the currency exchange service with the lookup service on the local machine, and use the found service to exchange one currency into another currency.

22.8 Modify Exercise 22.6 to add a set of Entry attributes to the service. The attributes should include the name of the exchange service, the address of the exchange service and any other attributes you want to add to the service. Use part or all of the attributes set to find a matching service.

BIBLIOGRAPHY
Objectives

- To be able to use the JavaSpaces service for building distributed applications.
- To understand the operations available in a JavaSpaces service.
- To be able to match entries in a JavaSpaces service against templates.
- To understand the use of transactions in JavaSpaces services.
- To be able to use notifications to build event-driven JavaSpaces applications.

The world is a book, and those who do not travel read only a page.
Saint Augustine

Write what you like; there is no other rule.
O. Henry

Take nothing on its looks: take everything on evidence.
There’s no better rule.
Charles Dickens

Believe nothing, no matter where you read it, or who said it, no matter if I have said it, unless it agrees with your own reason and your own common sense.
Buddha
Chapter 23  JavaSpaces  1259

23.1 Introduction

Objects that take part in distributed systems must be able to communicate with one another and share information. Thus far we have introduced several mechanisms by which Java objects in distributed systems can communicate. Java servlets (Chapter 9) enable Java objects (and non-Java objects) to communicate using the HTTP protocol. RMI (Chapter 13) enables Java objects running in separate virtual machines to invoke methods on one another as if those objects were in the same virtual machine. The Java Message Service (Chapter 16) enables Java objects (and non-Java objects) to communicate by publishing and consuming simple messages.

The JavaSpaces service is a Jini service that implements a simple, high-level architecture for building distributed systems. The JavaSpaces service enables Java objects to communicate, share objects and coordinate tasks using an area of shared memory. A JavaSpaces service provides three fundamental operations—write, take and read. The write operation places an object—called an entry—into the JavaSpaces service. The take opera-
tion specifies a template and removes from the JavaSpaces service an entry that matches the given template. The read operation is similar to the take operation, but does not remove the matching entry from the JavaSpaces service. In addition to the three basic operations, JavaSpaces services support transactions through the Jini transaction manager, and a notification mechanism that notifies an object when an entry that matches a given template is written to the JavaSpaces service.

In the first half of this chapter, we present fundamental JavaSpaces technology concepts and use simple examples to demonstrate operations, transactions and notifications. The case study at the end of this chapter uses JavaSpaces services to build a distributed image-processing application. This image-processing application uses JavaSpaces services to distribute the work of applying filters to images across many programs (normally on separate computers).

### 23.2 JavaSpaces Service Properties

JavaSpaces technology eases the design and development of distributed systems. A JavaSpaces service has five major properties:

1. A JavaSpaces service is a Jini service.
2. Multiple processes can access a JavaSpaces service concurrently.
3. An entry stored in a JavaSpaces service will remain in the JavaSpaces service until its lease expires or until a program takes the entry from the JavaSpaces service.
4. A JavaSpaces service locates objects by comparing those objects to a template. The template specifies the search criteria against which the JavaSpaces service compares each entry. When one or more entries match the template, the JavaSpaces service returns a single matching entry.
5. JavaSpaces services use the Jini transaction manager to ensure operations execute atomically.
6. Objects in a JavaSpaces service are shared. Programs can read and take entries from the JavaSpaces service, modify the public fields in those entries and write them back to the JavaSpaces service for other programs to use.

### 23.3 JavaSpaces Service

The JavaSpaces service provides distributed, shared storage for Java objects. Any Java-compatible client can put shared objects into the storage. However, several restrictions apply to these Java objects. First, any object stored in the JavaSpaces service must implement interface `Entry` (package `net.jini.core.entry`). JavaSpaces service `Entry`s adhere to the Jini `Entry` contract defined in the Jini Core Specification (see Chapter 22, Jini). An `Entry` can have multiple constructors and as many methods as required. Other requirements include a public no-argument constructor, public fields and no-primitive type fields. The JavaSpaces service proxy uses the no-argument constructor to instantiate the matching `Entry` during the deserialization process. All fields that will be used as the template matching fields in an `Entry` must be public (for more information on template matching fields, see Section 23.8). As defined by the Jini Core Specification, an `Entry` cannot have primitive-type fields. The object field requirement simplifies the model for template matching because primitive types cannot have null values, which are used as wildcards in templates.
JavaSpaces technology, like Jini, requires several underlying services. The JavaSpaces service depends on the Jini lookup service (for more information on Jini services, see Chapter 22, Jini). When transactions are required, the Jini transaction service (Section 23.11.2) must be started. JavaSpaces services also depend on a Web server and rmid (for more information on starting these services, see Chapter 22, Jini). Section 23.6 explains the relationships between JavaSpaces services and these Jini services. Section 23.11 demonstrates using the Transaction service with the JavaSpaces service. To use the JavaSpaces service, we need to start outrigger, which is Sun’s implementation of the JavaSpaces service. Two versions of the JavaSpaces service are available. One is the transient JavaSpaces service (nonactivatable). The other is the persistent JavaSpaces service (activatable). The transient JavaSpaces service does not require the RMI activation daemon (rmid), because the transient JavaSpaces service is not activatable. Once the transient JavaSpaces service terminates, all state information is lost and rmid is unable to restart the service. The persistent JavaSpaces service is activatable, and therefore requires the RMI activation daemon. If the persistent JavaSpaces service terminates, all of its state information is stored in a log file and rmid can restart the service at a later time.

To start the transient JavaSpaces service enter the following a command prompt

```
java -Djava.security.policy=policy
       -Djava.rmi.server.codebase=http://hostname:port/outrigger-dl.jar
       -jar c:\files\jini1_1\lib\transient-outrigger.jar public
```

where policy is the path to an appropriate policy file, hostname is the name of the machine on which the Web server is running and port specifies the port number on which the Web server will accept connections. The argument public specifies to which group this service belongs.

The following command starts the persistent JavaSpaces service:

```
java -jar c:\files\jini1_1\lib\outrigger.jar
       http://hostname:port/outrigger-dl.jar
       policy log_path public
```

where hostname is the name of the machine on which the Web server is running, port specifies the port number from which the Web server will accept connections, policy is the full path of the policy file and log_path is the location where the outrigger log will be created.

An additional parameter is useful in systems where two or more JavaSpaces services operate. The parameter is

```
-Dcom.sun.jini.outrigger.spaceName=name
```

where name defines the String with which the JavaSpaces service will register itself in the Jini lookup service. The default name for the JavaSpaces service is "JavaSpace". When searching for a specific JavaSpaces service in the Jini lookup service, you must use a Name Entry (package net.jini.lookup.entry) and initialize it to the String specified in the previous parameter. While this is a necessary parameter for systems with two or more JavaSpaces services, the examples in this chapter assume that only one JavaSpaces service exists in the system. The examples introduce a simple by which clients can locate the JavaSpaces service from the Jini lookup service.
You can start either a transient or persistent JavaSpaces service from the **StartService** GUI tool included in the Jini distributions. Start the GUI as in Chapter 21, then choose the **TransientSpace** tab (for a transient JavaSpaces service) or the **FrontEndSpace** (for a persistent JavaSpaces service). Go to the **Run** tab and click the **Start TransientSpace** button to run the transient service or click the **Start FrontEndSpace** button to run the persistent service.

### 23.4 Discovering the JavaSpaces Service

Upon initialization, each JavaSpaces service registers itself with local Jini lookup services. We assume that you already know how to start a Web server and RMI activation daemon from Chapter 22. The following is an example of a command that starts the persistent JavaSpaces service. Replace *hostname* with the name or IP address of your computer and *port* with the port number on which the Web server is listening.

```java
java -jar C:\files\jini1_1\lib\outtrigger.jar
    http://hostname:port/outtrigger-dl.jar
    C:\files\jini1_1\policy\policy.all
    C:\tmp\outtrigger_log public
```

Class **JavaSpaceFinder** (Fig. 23.1) shows how to obtain access to a JavaSpaces service. (We use this class in the example of Fig. 23.3.) The application performs unicast discovery to find the Jini lookup service on the hostname that the user specifies. Lines 35–36 get the **LookupLocator** at a user-specified Jini URL and obtain its **ServiceRegistrar**. Lines 55-68 look for all JavaSpaces services registered in the lookup service. Lines 55–57 specify a **ServiceTemplate** object (package `net.jini.core.lookup`). Lines 61–62 use the **ServiceTemplate** to search for all matching services in the lookup service and obtain a **JavaSpace**. Method **getJavaSpace** (lines 77–80) returns the discovered **JavaSpace**. For more information on how to use the Jini lookup service, please refer to Chapter 22, Jini.

```
// JavaSpaceFinder.java
// This application unicast discovers the JavaSpaces service.
package com.deitel.advjhtp1.javaspace.common;

// Jini core packages
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.*;
import net.jini.core.entry.Entry;

// Jini extension package
import net.jini.space.JavaSpace;

// Java core packages
import java.io.*;
import java.rmi.*;
import java.net.*;
```

**Fig. 23.1** Discovering a JavaSpaces service (part 1 of 3).
// Java extension package
import javax.swing.*;

public class JavaSpaceFinder {
    private JavaSpace space;

    public JavaSpaceFinder( String jiniURL )
    {
        LookupLocator locator = null;
        ServiceRegistrar registrar = null;

        System.setSecurityManager( new RMISecurityManager() );

        try {
            locator = new LookupLocator( jiniURL );
            registrar = locator.getRegistrar();
        }

        catch ( MalformedURLException malformedURLException ) {
            malformedURLException.printStackTrace();
        }

        catch ( java.io.IOException ioException ) {
            ioException.printStackTrace();
        }

        catch ( ClassNotFoundException classNotFoundException ) {
            classNotFoundException.printStackTrace();
        }

        // specify the service requirement
        Class[] types = new Class[] { JavaSpace.class };
        ServiceTemplate template =
            new ServiceTemplate( null, types, null );

        // find service
        try {
            space =
                ( JavaSpace ) registrar.lookup( template );
        }

        catch ( RemoteException remoteException ) {
            remoteException.printStackTrace();
        }
Clients access objects in a JavaSpaces service through interface `JavaSpace` (package `net.jini.space`). Interface `JavaSpace` provides several methods—`notify`, `read`, `readIfExists`, `take`, `takeIfExists`, `write` and `snapshot`. The purpose of each method is as follows:

1. `write`—This method implements the `write` operation. The `write` operation inserts an `Entry` into a JavaSpaces service. If an identical `Entry` already exists in the JavaSpaces service, this operation does not overwrite the existing `Entry`. Instead, the `write` operation places a copy of the `Entry` into the JavaSpaces service. JavaSpaces services may contain multiple copies of the same `Entry`. Section 23.7 demonstrates how to use the `write` operation.

2. `read`, `readIfExists`—These two methods implement the `read` operation, which attempts to read an `Entry` that matches an `Entry` template from a JavaSpaces service. If no matching `Entry` exists in the JavaSpaces service, this operation returns `null`. If multiple matching `Entry` exist in the JavaSpaces service, the `read` operation arbitrarily picks one among the matching `Entries`. Method `read` blocks until a matching `Entry` is found in the JavaSpaces service or until a time-out occurs. Method `readIfExists` checks to see if a matching `Entry` exists within the JavaSpaces service. If an `Entry` does not exist in the JavaSpaces service, method `readIfExists` should return `null` immediately. Method `readIfExists` does not block unless the matching `Entry` is a participant in an uncommitted transaction. For information on transactions, see Section 23.11. Section 23.8.1 demonstrates the `read` and `readIfExists` operations.

3. `take`, `takeIfExists`—These two methods implement the `take` operation, which attempts to remove an `Entry` that matches an `Entry` template from a JavaSpaces service. This operation works like the `read` operation, except that the `take` operation removes the matching `Entry` from the JavaSpaces service. Method `take` blocks until a matching `Entry` is found in the JavaSpaces service or until a time-out occurs. Method `takeIfExists` checks to see if a matching `Entry` exists within the JavaSpaces service. If an `Entry` does not exist in the JavaSpaces service, method `takeIfExists` should return `null` immediately.
Method `takeIfExists` does not block unless the matching `Entry` is part of an uncommitted transaction. Section 23.8.2 demonstrates the `take` and `takeIfExists` operations.

4. `notify`—This method implements the `notify` operation, which requests that the JavaSpaces service sends a notification to a listener object when a client writes a matching `Entry` into the JavaSpaces service. With this method, an application does not need to check repeatedly for an `Entry` in a JavaSpaces service. Section 23.9 demonstrates the `notify` operation.

5. `snapshot`—This method increases performance when a program must serialize one `Entry` repeatedly. Each time a program transfers an `Entry` into a JavaSpaces service (e.g., by writing that `Entry` or by using that `Entry` as a template), that `Entry` must be serialized. If a program transfers the same `Entry` to a JavaSpaces service many times, the serialization process can be time consuming. Invoking method `snapshot` serializes the `Entry` once and reuses this serialized `Entry` for future transfers. Section 23.10 demonstrates how to use method `snapshot`.

### 23.6 Defining an Entry

The following sections and subsections create an application for managing registrations for fictitious seminars offered at Deitel & Associates, Inc. For each seminar, an administration tool writes an `AttendeeCounter` (Fig. 23.2) into the JavaSpaces service. Each `AttendeeCounter` keeps track of the number of attendees registered for a particular seminar. We implement the complete application one step at a time to demonstrate the mechanics of each of each JavaSpaces service operation.

`AttendeeCounter` (Fig. 23.2) is an `Entry` that represents a count of the number of people attending a seminar. Recall that `Entries` require non-primitive type, `public` fields (lines 9–10) and an empty constructor (line 13). The `AttendeeCounter` constructor on lines 16–19 takes as a `String` argument the day of the week for which this `AttendeeCounter` tracks attendee registrations.

**Common Programming Error 23.1**

Including primitive fields in an `Entry` does not cause an error during compilation. However, an `IllegalArgumentException` does occur during serialization.

```java
// Fig. 23.2: AttendeeCounter.java
// This class defines the AttendeeCounter Entry.
package com.deitel.advjhtp1.javaspace.common;

import net.jini.core.entry.Entry;

public class AttendeeCounter implements Entry {

    public String day;
    public Integer counter;
}
```

Fig. 23.2 `AttendeeCounter` is an `Entry` for keeping track of registrations for a seminar on a particular day (part 1 of 2).
Software Engineering Observation 23.1

Use wrapper classes rather than primitive types in Entry fields.

23.7 Write Operation

The write operation places an Entry in a JavaSpaces service. Method write takes three arguments—an Entry, a Transaction object and a long value that requests an amount of time for which the JavaSpaces service should keep the Entry. The long value represents the lease length for the Entry. Normally, the JavaSpaces service grants each written Entry a lease time of 5 minutes. The JavaSpaces service will not keep the Entry beyond the lease time granted. A developer can extend the life of an Entry by renewing its lease before it expires. Method write returns a net.jini.lease.Lease object that contains the time that the JavaSpaces service granted to the Entry. Method write throws two exceptions. Method write throws a RemoteException (package java.rmi) when a network failure occurs or a variety of other errors occur on the server. When a write operation takes place under an invalid transaction, method write throws a TransactionException (package net.jini.core.transaction). For information on transactions, see Section 23.11.

The WriteOperation application (Fig. 23.3) uses class AttendeeCounter (Fig. 23.2) and class JavaSpaceFinder (Fig. 23.1) to demonstrate writing an Entry into a JavaSpaces service. In this example, a seminar administrator would use the WriteOperation application to place AttendeeCounter Entries for each available seminar in the JavaSpaces service. The constructor (lines 30–36) takes a JavaSpace as an argument. Method writeEntry (lines 39–62) writes an Entry into the JavaSpaces service. Lines 44–45 initialize an Entry by setting the number of people who register for the seminar to zero. Line 46 writes the Entry into the JavaSpaces service. The first argument (counter) specifies the Entry to write into the JavaSpaces service. The second argument (null) indicates that the write operation does not use a Transaction. When the write operation completes, the written Entry is ready for a read or take operation. If a Transaction object is specified, the write operation uses that Transaction to ensure that a series of operations completes successfully. This means that until the transaction completes successfully, other clients cannot read or take the Entry from the JavaSpaces service. The third argument (Lease.FOREVER) specifies how long the JavaSpaces service should keep the Entry. Although we request that the JavaSpaces service keep our Entry forever, Sun’s implementation limits the lease to 5 minutes. Pro-
grams can use the Jini lease-renewal mechanism to maintain leases for entries. After the lease expires, the JavaSpaces service removes and destroys the object.

Method `showOutput` (lines 65–75) displays the results. In method `main`, lines 81–85 check the user-specified hostname. Lines 88–90 ask user to choose a particular day to write. Figure 23.4 shows the results of running the `WriteOperation` application.

```java
// WriteOperation.java
// This application initializes an new Entry, and puts this Entry to the JavaSpace.
package com.deitel.advjhtp1.javaspace.write;

// Jini core packages
import net.jini.core.lease.Lease;
import net.jini.core.transaction.TransactionException;

// Jini extension package
import net.jini.space.JavaSpace;

// Java core package
import java.rmi.RemoteException;

// Java extension package
import javax.swing.*;

// Deitel package
import com.deitel.advjhtp1.javaspace.common.*;

public class WriteOperation {
    private JavaSpace space;
    private static final String[] days = { "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday" };  
    private String output = "\n";

    // WriteOperation constructor
    public WriteOperation( String hostname ) {
      // get JavaSpace
      String jiniURL = "jini://" + hostname;
      JavaSpaceFinder findtool = new JavaSpaceFinder( jiniURL );
      space = findtool.getJavaSpace();
    }

    // deposit new Entry to JavaSpace
    public void writeEntry( String day ) {
      // initialize AttendeeCounter Entry and deposit
      // Entry in JavaSpace
      try {
        AttendeeCounter counter = new AttendeeCounter( day );
        space.write( counter, null, Lease.FOREVER );
      }
    }
}
```

Fig. 23.3 Writing an Entry into a JavaSpaces service (part 1 of 2).
output += "Initialize the Entry: \n";
output += "   Day: " + day + "\n";
output += "   Count: 0\n";
}
// handle exception network failure
catch ( RemoteException exception ) {
    exception.printStackTrace();
}
// handle exception invalid transaction
catch ( TransactionException exception ) {
    exception.printStackTrace();
}
}
// show output
public void showOutput()
{
    JTextArea outputArea = new JTextArea();
    outputArea.setText( output );
    JOptionPane.showMessageDialog( null, outputArea,
    "WriteOperation Output",
    JOptionPane.INFORMATION_MESSAGE );
    // terminate program
    System.exit( 0 );
}
// method main
public static void main( String args[] )
{
    // get hostname
    if ( args.length != 1 ) {
        System.out.println("Usage: WriteOperation hostname");
        System.exit( 1 );
    }
    // get user input day
    String day = ( String ) JOptionPane.showInputDialog(null, "Select Day", "Day Selection",
    JOptionPane.QUESTION_MESSAGE, null, days, days[ 0 ] );
    // write Entry
    WriteOperation write = new WriteOperation( args[ 0 ] );
    write.writeEntry( day );
    write.showOutput();
} // end method main

Fig. 23.3 Writing an Entry into a JavaSpaces service (part 2 of 2).
This application takes a command-line argument that specifies the hostname of a computer that has the JavaSpaces service running. The following steps compile and execute the WriteOperation application. Ensure that your CLASSPATH includes jini-core.jar, jini-ext.jar and sun-util.jar. Compile the java files in the com\deitel\advjhtp1\javaspace\common directory. Run WriteOperation by specifying the hostname of the Jini lookup service. Do not forget to specify to the JVM the policy file with the proper permissions.

23.8 Read and Take Operations

The read and the take operations retrieve Entries from a JavaSpaces service. A client can read or take an Entry from the JavaSpaces service by supplying a template Entry against which to compare the public fields of Entries in the JavaSpaces service. The template indicates which fields to use for comparison purposes.

The retrieval process uses a template-matching mechanism to match Entries according to the values of their public fields. Each Entry in the JavaSpaces service requires its public fields to be object references, so each field is either null or a reference to an object. Fields in the template with non-null values must match with their Entry counterparts in the JavaSpaces service exactly. Fields in the template that are set to null act as wildcards. If a set of Entries of the same type exist within a JavaSpaces service, only those fields which equal those of the template are used to match an Entry or a set of Entries contained in the JavaSpaces service. Fields in the template set to null can have their matching counterparts in the JavaSpaces service have any value in the corresponding field(s).

23.8.1 Read Operation

The read operation obtains Entries without removing them from the JavaSpaces service. Methods read and readIfExists perform the read operation. Each method takes three arguments—an Entry that specifies the template to match, a Transaction object and a long value. The long value has different meanings in methods read and readIfExists. Method read specifies a period of time for which the read operation should block before simply returning null. Method readIfExists is a non-blocking version of method read. If there are no matching Entries, readIfExists returns null immediately. Method readIfExists blocks only if the developer specifies a period of time for which the readIfExists waits if, at first, the matching Entry is a part of an incomplete transaction. If the matching Entry is not involved in any transaction, then the read operation returns the matching Entry immediately. Both the read and readIfExists methods re-
turn only one **Entry**. If multiple **Entry**s match the template, the **read** operation picks one arbitrarily. These methods throw four exception types—`RemoteException`, `TransactionException`, `UnusableEntryException` and `InterruptedException`. The first two are the same as in method **write**. If the matching **Entry** cannot be deserialized, these methods throw an **UnusableEntryException** (package net.jini.core.entry).

The **ReadOperation** application (Fig. 23.5) uses class **AttendeeCounter** (Fig. 23.2) and class **JavaSpaceFinder** (Fig. 23.1) to demonstrate reading an **Entry** from a JavaSpaces service. A seminar administrator or prospective attendee could use this application to determine the current enrollment for a particular seminar. Line 47 specifies the matching template against which to compare **Entries**. Users must specify the day for which they would like to see attendee registrations.

```java
// ReadOperation.java
// This application reads an Entry from the JavaSpace and displays the Entry information.
package com.deitel.advjhtp1.javaspace.read;

// Jini core packages
import net.jini.core.transaction.TransactionException;
import net.jini.core.entry.UnusableEntryException;

// Jini extension package
import net.jini.space.JavaSpace;

// Java core packages
import java.rmi.RemoteException;
import java.lang.InterruptedException;

// Java extension package
import javax.swing.*;

// Deitel package
import com.deitel.advjhtp1.javaspace.common.*;

public class ReadOperation {

    private JavaSpace space;

    private static final String[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };

    private String output = "\n";

    // constructor gets JavaSpace
    public ReadOperation( String hostname )
    {
        // get JavaSpace
        String jiniURL = "jini://" + hostname;
        JavaSpaceFinder findtool = new JavaSpaceFinder( jiniURL );
        space = findtool.getJavaSpace();
    }
```

**Fig. 23.5** Reading an Entry from JavaSpaces service (part 1 of 3).
public void readEntry( String day )
{
    // specify matching template, read template
    // from JavaSpace and output Entry information
    try {

        // read Entry from JavaSpace
        AttendeeCounter counter = new AttendeeCounter( day );
        AttendeeCounter resultCounter = ( AttendeeCounter )
            space.read( counter, null, JavaSpace.NO_WAIT );

        if (resultCounter == null) {
            output += "Sorry, cannot find an Entry for "
                    + day + "!\n";
        } else {

            // get Entry information
            output += "Count Information:\n";
            output += " Day: " + resultCounter.day;
            output += "\n";
            output += " Count: "
                    + resultCounter.counter.intValue() + "\n";
        }
    }

    // handle exception network failure
    catch ( RemoteException exception ) {
        exception.printStackTrace();
    }

    // handle exception invalid transaction
    catch ( TransactionException exception ) {
        exception.printStackTrace();
    }

    // handle exception unusable Entry
    catch ( UnusableEntryException exception ) {
        exception.printStackTrace();
    }

    // handle exception interrupting
    catch ( InterruptedException exception ) {
        exception.printStackTrace();
    }

    // end method readEntry
}

// show output
public void showOutput()
{
    JTextArea outputArea = new JTextArea();
The first argument to method `read` (lines 48–49) specifies an `Entry` that the template-matching mechanism will use. The second argument (`null`) indicates that this `read` operation does not use a `Transaction`. The third argument (`JavaSpace.NO_WAIT`) specifies the period for which the read operations wait for the read operation to find a matching `Entry` before simply returning `null`. Our example sets the method `read` to `JavaSpace.NO_WAIT`, which equals zero. If the template-matching mechanism does not locate a matching `Entry`, the `read` operation returns `null` immediately. Figure 23.6 shows the results of running the `ReadOperation` application.

This application takes a command-line argument that specifies the hostname of a machine that has the JavaSpaces service service running. The following steps compile and execute the `ReadOperation` application. Ensure that your `CLASSPATH` includes `jini-core.jar`, `jini-ext.jar` and `sun-util.jar`. Compile the java files in the `com\deitel\advjhtp1\javaspace\read` directory. Run `ReadOperation` by specifying the hostname of the Jini lookup service. Do not forget to specify to the JVM the policy file with the proper permissions.
Software Engineering Observation 23.2

The read operation returns only a single matching Entry. If multiple matching Entries exist, the read operation may return different matching objects each time.

23.8.2 Take Operation

The take operation obtains an Entry and removes it from the JavaSpaces service. Methods take and takeIfExists perform the take operation. Methods take and takeIfExists are similar to methods read and readIfExists. The only difference is that the matching Entry returned by a take or takeIfExists operation is removed from the JavaSpaces service.

The TakeOperation application (Fig. 23.7) uses class AttendeeCounter (Fig. 23.2) and class JavaSpaceFinder (Fig. 23.1) to demonstrate taking an Entry from a JavaSpaces service. This application is similar to the ReadOperation application. The only difference is that this application calls method take (lines 46–47) of interface JavaSpace to remove the AttendeeCounter from the JavaSpaces service. A seminar administrator could use this application to remove from the JavaSpaces service an AttendeeCounter for a seminar that has already been given or for a seminar that was cancelled. Figure 23.8 shows the results of running the TakeOperation application.

This application takes a command-line argument that specifies the hostname of a machine that has the JavaSpaces service running. The following steps compile and execute the TakeOperation application. Ensure that your CLASSPATH includes jini-core.jar, jini-ext.jar and sun-util.jar. Compile the java files in the com\deitel\advjhtpl\javaspace\take directory. Run TakeOperation by specifying the hostname of the Jini lookup service. Do not forget to specify to the JVM the policy file with the proper permissions.

```java
// TakeOperation.java
// This application removes an Entry from the JavaSpace.
package com.deitel.advjhtpl.javaspace.take;

// Jini core packages
import net.jini.core.transaction.TransactionException;
import net.jini.core.entry.UnusableEntryException;
```

Fig. 23.7 Taking an Entry from a JavaSpaces service (part 1 of 4).
public class TakeOperation {

    private JavaSpace space = null;
    private static final String[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
    private String output = "\n";

    public TakeOperation( String hostname ) {
        // get JavaSpace
        String jiniURL = "jini://" + hostname;
        JavaSpaceFinder findtool = new JavaSpaceFinder( jiniURL );
        space = findtool.getJavaSpace();
    }

    // remove Entry from JavaSpace
    public void TakeAnEntry( String day ) {
        AttendeeCounter resultCounter = null;
        // specify matching template, remove template
        // from JavaSpace
        try {
            AttendeeCounter count = new AttendeeCounter( day );
            resultCounter = ( AttendeeCounter ) space.take( count, null, JavaSpace.NO_WAIT );

            if ( resultCounter == null ) {
                output += "No Entry for " + day + " is available from the JavaSpace.\n";
            } else {
                output += "Entry is taken away from ";
                output += "the JavaSpace successfully.\n";
            }
        }
    }
}
// handle exception network failure
try {
    exception.printStackTrace();
} catch ( RemoteException exception ) {
    
    // handle exception invalid transaction
    catch ( TransactionException exception ) {
        exception.printStackTrace();
    } catch ( UnusableEntryException exception ) {
        exception.printStackTrace();
    } catch ( InterruptedException exception ) {
        exception.printStackTrace();
    }
}

// show output
public void showOutput()
{
    JTextArea outputArea = new JTextArea();
    outputArea.setText(output);
    JOptionPane.showMessageDialog(null, outputArea, 
        "TakeOperation Output", 
        JOptionPane.INFORMATION_MESSAGE);
}

// terminate program
System.exit(0);

public static void main(String args[])
{
    // get hostname
    if ( args.length != 1 ) {
        System.out.println("Usage: WriteOperation hostname");
        System.exit(1);
    }

    // get user input day
    String day = (String) JOptionPane.showInputDialog(null, "Select Day", "Day Selection", 
        JOptionPane.QUESTION_MESSAGE, null, days, days[0]);

    // take Entry
    TakeOperation take = new TakeOperation(args[0]);
    take.TakeAnEntry(day);

Fig. 23.7  Taking an Entry from a JavaSpaces service (part 3 of 4).
Software Engineering Observation 23.3

The take operation returns only a single matching Entry. If multiple matching Entries exist, the take operation can remove only one Entry from the JavaSpaces service each time. To take all the matching Entries away from the JavaSpaces service, execute the TakeOperation application repeatedly until the TakeOperation application returns a message that says “No Entry is available from the JavaSpaces service.”

23.9 Notify Operation

The notify operation asks the JavaSpaces service to send a notification to a listener when a client writes a matching Entry into the JavaSpaces service. Method notify takes five parameters—an Entry that specifies the matching template, a Transaction object, a listener that implements interface RemoteEventListener (package net.jini.core.event), a long value that specifies the lease time for the registration of the listener and a MarshalledObject (package java.rmi) that the JavaSpaces service will pass to the remote listener as part of a notification. This method may throw exceptions of type RemoteException and TransactionException. A RemoteException occurs due to a network failure. A TransactionException occurs when a notify operation takes place as part of an invalid transaction.

Class EntryListener (Fig. 23.9) defines a listener that the JavaSpaces service will notify when an Entry matching the given template is written to the JavaSpaces service. The EntryListener listens on the JavaSpaces service for a matching Entry written into the JavaSpaces service. This listener must implement interface RemoteEventListener (line 14). A person interested in attending a seminar on a particular day could use this application to be notified when an AttendeeCounter is added for a seminar on a particular day. The constructor takes one argument—a RemoteEventListener and exports the listener to the JavaSpaces service so that when a client writes a matching Entry into the JavaSpaces service, the JavaSpaces service will call notify. Method notify (lines 35–40) forwards the notification to the NotifyOperation application (Fig. 23.10).
Chapter 23 JavaSpaces

The **NotifyOperation** application (Fig. 23.10) demonstrates how to write a program that receives a notification when a matching **Entry** is written into a JavaSpaces service. Lines 30–36 define the constructor, which gets a JavaSpaces service. In method **notifyEntry** (lines 39–61), line 42 gets an **EntryListener** that listens on the JavaSpaces service for matching **Entry**s. This **EntryListener** will be passed to method **notify** of interface **JavaSpace**. Line 48 creates the matching template. Lines 50–51 define the object to send to the listener when a notification occurs. Lines 52–53 call the **notify** method of the **JavaSpace** interface. The first argument (**counter**) specifies an **Entry** that is used as a matching template. The second argument (**null**) indicates that
the `notify` operation does not occur within a transaction. The third argument (listener) is an instance of the `EntryListener` class. The fourth argument (600000) specifies the number of milliseconds requested for the lease. After the expiration of the listener’s granted lease, the listener will cease to be active. The last argument (handback—a reference to a `MarshalledObject`) is an object that the JavaSpaces service provides to the remote listener as part of the notification.

```java
// NotifyOperation.java
// This application receives a notification when a matching entry
// is written to the JavaSpace.
package com.deitel.advjhtp1.javaspace.notify;

// Jini core packages
import net.jini.core.transaction.TransactionException;
import net.jini.core.lease.Lease;
import net.jini.core.event.*;

// Jini extension package
import net.jini.space.JavaSpace;

// Java core packages
import java.rmi.*;

// Java standard extensions
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.javaspace.common.*;

public class NotifyOperation implements RemoteEventListener
{
    private JavaSpace space;
    private static final String[] days = { "Monday", "Tuesday",
                                         "Wednesday", "Thursday", "Friday" };

    // constructor gets JavaSpace
    public NotifyOperation( String hostname )
    {
        // get JavaSpace
        String jiniURL = "jini://" + hostname;
        JavaSpaceFinder findtool = new JavaSpaceFinder( jiniURL );
        space = findtool.getJavaSpace();
    }

    // call notify method of JavaSpace
    public void notifyEntry( String day )
    {
        // get Entry listener
        EntryEventListener listener = new EntryListener( this );
    }
}
```

*Fig. 23.10* Receiving notifications when matching Entrys are written into JavaSpace (part 1 of 3).
44 // specify matching template, asks JavaSpace to
45 // send notification when matching entry is written
46 // to JavaSpace
47 try {
48     AttendeeCounter counter = new AttendeeCounter( day );
49     
50     MarshalledObject handback = new MarshalledObject(  
51             "JavaSpace Notification" );
52     space.notify(  
53             counter, null, listener, 10 * 60 * 1000, handback );
54 }
55
56 // handle exception notifying space
57 catch ( Exception exception ) {
58     exception.printStackTrace();
59 }
60
61 } // end method notifyEntry
62
63 // show output
64 public void showOutput( String output )
65 {
66     JTextArea outputArea = new JTextArea();
67     outputArea.setText( output );
68     JOptionPane.showMessageDialog( null, outputArea,  
69             "NotifyOperation Output",  
70             JOptionPane.INFORMATION_MESSAGE );
71 }
72
73 // receive notifications
74 public void notify( RemoteEvent remoteEvent )
75 {
76     String output = "\n";
77     
78     // prepare output
79     try {
80         output += "id: " + remoteEvent.getID() + "\n";  
81         output += "sequence number: "  
82             + remoteEvent.getSequenceNumber() + "\n";
83         String handback = ( String )  
84             remoteEvent.getRegistrationObject().get();
85         output += "handback: " + handback + "\n";
86     
87     // display output
88     showOutput( output );
89 }
90
91 // handle exception getting handback
92 catch ( Exception exception ) {
93     exception.printStackTrace();
94 }
95 }

Fig. 23.10 Receiving notifications when matching Entrys are written into JavaSpace (part 2 of 3).
The following steps execute the `NotifyOperation` application. Ensure that your `CLASSPATH` includes `jini-core.jar`, `jini-ext.jar` and `sun-util.jar`. Compile the source files in the `com\deitel\advjhtp1\javaspace\notify` directory. Start a Web server. Generate a stub for class `EntryListener` (Fig. 23.9). Create a JAR file for `EntryListenerStub.class` and place it in the Web server’s document directory. Run `NotifyOperation` by specifying the hostname of the Jini lookup service. Do not forget to specify to the JVM the codebase and the policy file with the proper permissions.

Figure 23.11 shows sample outputs of this application. To test this application, execute several `WriteOperation` applications.

```java
// method main
public static void main( String args[] )
{
    // get hostname
    if ( args.length != 1 ) {
        System.out.println(
            "Usage: NotifyOperation hostname" );
        System.exit( 1 );
    }
    // get user input day
    String day = ( String ) JOptionPane.showInputDialog(
        null, "Select Day", "Day Selection",
        JOptionPane.QUESTION_MESSAGE, null, days, days[ 0 ] );
    // notify Entry
    NotifyOperation notifyOperation =
        new NotifyOperation( args[ 0 ] );
    notifyOperation.notifyEntry( day );
} // end method main
```

Fig. 23.10 Receiving notifications when matching `Entry`s are written into `JavaSpace` (part 3 of 3).

Fig. 23.11 `NotifyOperation` Output samples.
Software Engineering Observation 23.4

JavaSpaces service notifications are not guaranteed to be delivered, as network problems may interfere with notification delivery.

23.10 Method **snapshot**

Method **snapshot** optimizes interactions with a JavaSpaces service by reducing the overhead of continually serializing **Entry**s. Every time we pass a matching template to methods in the **JavaSpace** interface, the template must be serialized before it is moved to the JavaSpaces service. When the same template is passed to a JavaSpaces service repeatedly, it is preferable to avoid multiple serializations of the same **Entry**. Fortunately, method **snapshot** provides such a mechanism. Method **snapshot** takes a matching template and returns a specialized representation of the **Entry** (a **snapshot** **Entry**). This snapshot **Entry** can be used only in the JavaSpaces service that generated it. For example, to remove all the seminar **Entry**s for Monday from the JavaSpaces service, we call the **snapshot** method to create a snapshot **Entry**, then pass this snapshot **Entry** to the **take** method repeatedly.

The **SnapshotUsage** application (Fig. 23.12) removes **Entry**s from the JavaSpaces service and uses method **snapshot** to avoid repeated serialization of the matching template. Line 49 defines the original matching template. We do not pass this original template to method **take**. Instead, we pass the snapshot to the **take** method. Line 50 calls method **snapshot** to get the snapshot **Entry** of the original template.

```java
// SnapshotUsage.java
// This application removes entries from the JavaSpace using method snapshot.
package com.deitel.advjhtp1.javaspace.snapshot;

// Jini core packages
import net.jini.core.transaction.TransactionException;
import net.jini.core.entry.UnusableEntryException;
import net.jini.core.entry.Entry;

// Jini extension package
import net.jini.space.JavaSpace;

// Java core packages
import java.rmi.RemoteException;

// Java extension package
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.javaspace.common.*;

public class SnapshotUsage {
    private JavaSpace space;
}
```

Fig. 23.12 Removing entries from JavaSpaces service using method **snapshot** (part 1 of 3).
private static final String[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
private String output = "\n";

// constructor gets JavaSpace
public SnapshotUsage( String hostname ) {
    // get JavaSpace
    String jiniURL = "jini://" + hostname;
    JavaSpaceFinder findtool = new JavaSpaceFinder( jiniURL );
    space = findtool.getJavaSpace();
}

// create snapshot Entry, pass this object as
// Entry parameter to take method
public void snapshotEntry( String day ) {
    // specify matching template, snapshot template
    // and remove matching entries from JavaSpace using
    // snapshot entry
    try {
        AttendeeCounter counter = new AttendeeCounter( day );
        Entry snapshotentry = space.snapshot( counter );
        AttendeeCounter resultCounter = ( AttendeeCounter )
            space.take( snapshotentry, null, JavaSpace.NO_WAIT );

        // keep removing entries until no more entry exists
        // in space
        while ( resultCounter != null ) {
            output += "Removing an entry ... \n";
            resultCounter = ( AttendeeCounter ) space.take( snapshotentry, null, JavaSpace.NO_WAIT );
        }

        output += "No more entry to remove!\n";
    }

    // handle exception network failure
    catch ( RemoteException remoteException ) {
        remoteException.printStackTrace();
    }

    // handle exception invalid transaction
    catch ( TransactionException transactionException ) {
        transactionException.printStackTrace();
    }

    // handle exception unusable entry
    catch ( UnusableEntryException unusableEntryException ) {
        unusableEntryException.printStackTrace();
    }
}

Fig. 23.12 Removing entries from JavaSpaces service using method snapshot (part 2 of 3).
The only argument of method `snapshot` (line 48) specifies the template to serialize. Method `snapshot` returns a snapshot `Entry` that represents the matching template. Lines 49–51 call method `take` to remove the `Entry`s that match the template from the JavaSpaces service. Lines 54–58 remove all matching `Entry`s in the JavaSpaces service. Figure 23.13 shows the output of running the `SnapshotUsage` application. This output indicates that there are three matching `Entry`s in the JavaSpaces service. The `take` operation removes all three matching `Entry`s from the JavaSpaces service.
The following steps compile and execute the `SnapshotUsage` application. Make sure your `CLASSPATH` includes `jini-core.jar`, `jini-ext.jar` and `sun-util.jar`. Compile the java files in the `com\deitel\advjhtp1\javaspace\snapshot` directory. Run `SnapshotUsage` by specifying the hostname of the Jini lookup service. Do not forget to specify to the JVM the policy file with the proper permissions.

**Software Engineering Observation 23.5**

Using the snapshot `Entry` is equivalent to using the original `Entry`, as long as all operations take place on the same JavaSpaces service that generated the snapshot.

### 23.11 Updating Entries with Jini Transaction Service

We cannot modify an `Entry` in a JavaSpaces service directly. Instead, we must take the `Entry` away from the JavaSpaces service, change the values of the `Entry` fields, then place the `Entry` back into the JavaSpaces service. To ensure that the JavaSpaces service does not lose the `Entry` when a process takes that `Entry` away, we can perform the take, update and write processes in a transaction. If all these processes succeed, the transaction completes. Otherwise, the transaction fails and the JavaSpaces service returns the `Entry` to its state prior to the transaction.

Assume a distributed system in which dedicated nodes take `Entries` from a JavaSpaces service, process each `Entry`, and write them back to the JavaSpaces service when finished. What happens if a problem occurs and one of the dedicated nodes never returns a processed `Entry`? The information that node processed could be permanently lost. Furthermore, because the processing node removed the `Entry` from the JavaSpaces service, the unprocessed `Entry` is also lost. The use of a transaction manager protects a JavaSpaces service from these situations. When a transaction fails, the transaction manager restores the `Entry` to its previous state—as if the client never took the `Entry`.

Our next example demonstrates how to update an `Entry` in a JavaSpaces service. The application takes an `AttendeeCounter Entry` from the JavaSpaces service, updates the count variable and reinserts the `Entry` back into the JavaSpaces service. A seminar administrator could use this application to register a new attendee for a seminar and update the appropriate `AttendeeCounter`. We have to ensure that if a client takes an `Entry` from the JavaSpaces service, it will write the `Entry` back into the JavaSpaces service later. In this example, we use the Jini transaction manager to guarantee that only one client at a time can update a seminar `Entry`. Otherwise, one client potentially could overwrite a previously written `Entry`, thus corrupting the proper count of the number of people who will attend a seminar.
23.11.1 Defining the User Interface

This section defines the user interface for the application. To update an Entry, the program must know which AttendeeCounter to update and the number to add to the counter. Class UpdateInputWindow (Fig. 23.14) prompts the user for the day of the week to update and the number of people who will attend the seminar on that day.

```java
// UpdateInputWindow.java
// This application is an user interface used to get the input data.
package com.deitel.advjhtp1.javaspace.update;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class UpdateInputWindow extends JFrame {
    private String[] dates = { "Monday", "Tuesday",
                               "Wednesday", "Thursday", "Friday" };
    private JButton okButton;
    private JComboBox dateComboBox;
    private JLabel firstLabel;
    private JTextField numberText;
    private String date = "Monday";
    private String hostname;

    public UpdateInputWindow( String name ) {
        super( "UpdateInputWindow" );
        Container container = getContentPane();

        hostname = name;

        // define center panel
        JPanel centerPanel = new JPanel();
        centerPanel.setLayout( new GridLayout( 2, 2, 0, 5 ) );

        // add label
        firstLabel = new JLabel( "Please choose a date:",
                                   SwingConstants.CENTER );
        centerPanel.add( firstLabel );

        // add combo box
        dateComboBox = new JComboBox( dates );
        dateComboBox.setSelectedIndex( 0 );
        centerPanel.add( dateComboBox );
    }
}
```

Fig. 23.14 UpdateInputWindow user interface (part 1 of 3).
// install listener to combo box
setAttributeComboBox.addItemListener(
    new ItemListener() {
        public void itemStateChanged( ItemEvent itemEvent ) {
            date = ( String )setAttributeComboBox.getSelectedItem();
        }
    });

// add label
JLabel numberLabel = new JLabel(
    "Please specify a number:", SwingConstants.CENTER );
centerPanel.add( numberLabel );

// add text field
setAttributeText = new JTextField( 10 );
centerPanel.add( numberText );

// install listener to text field
numberText.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            count = Integer.parseInt( numberText.getText() );
        }
    });

// define button panel
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout( new GridLayout( 1, 1, 0, 5 ) );

// add OK button
okButton = new JButton( "OK" );
buttonPanel.add( okButton );

// add listener to OK button
okButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // get user input
            count = Integer.parseInt( numberText.getText() );
        }
    });

Fig. 23.14 UpdateInputWindow user interface (part 2 of 3).
23.11.2 Discovering the TransactionManager Service

Creating a transaction, requires a transaction manager. In our example, we use Jini’s TransactionManager service to obtain a transaction manager. We assume that you already know how to start the Web server and the RMI activation daemon from Chapter 22.

```
java -jar
-Dcom.sun.jini.mahalo.managerName=TransactionManager
c:\files\jini1_1\lib\mahalo.jar
http://hostname:port/mahalo-dl.jar
c:\files\jini1_1\policy\policy.all
c:\mahalo\txn_log public
```

Class TransactionManagerFinder (Fig. 23.15) demonstrates the TransactionManager service. This application performs unicast discovery to find the Jini lookup service, through which we can get a reference to the TransactionManager.  

The application looks for a TransactionManager in the lookup service. Lines 40–43 specify a ServiceTemplate object with which lines 46–47 search the lookup service. Method getTransactionManager (lines 72–75) returns a Transaction Manager.
// TransactionManagerFinder.java
// This application unicast discovers the
// TransactionManager service.
package com.deitel.advjhtp1.javaspace.common;

// Jini core packages
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.*;
import net.jini.core.entry.Entry;
import net.jini.core.transaction.server.TransactionManager;

// Java core packages
import java.io.*;
import java.rmi.RMISecurityManager;
import java.net.*;

// Java extension package
import javax.swing.*;

public class TransactionManagerFinder {

    private TransactionManager transactionManager = null;

    public TransactionManagerFinder( String jiniURL ) {
        LookupLocator locator = null;
        ServiceRegistrar registrar = null;

        System.setSecurityManager( new RMISecurityManager() );

        // get lookup service locator at "jini://hostname"
        // use default port
        try {
            locator = new LookupLocator( jiniURL );

            // get registrar for the locator
            registrar = locator.getRegistrar();

            // specify service requirement
            Class[] types = new Class[] {
                TransactionManager.class
            };
            ServiceTemplate template =
                new ServiceTemplate( null, types, null );

            // find service
            transactionManager =
                (TransactionManager) registrar.lookup( template );
        }

        // handle exception invalid jini URL
        catch ( MalformedURLException malformedURLException ) {
            malformedURLException.printStackTrace();
        }
    }
}
23.11.3 Updating an Entry

We now have the user interface (class `UpdateInputWindow`), the transaction manager (class `TransactionManagerFinder`) and the JavaSpaces service (class `JavaSpaceFinder`) ready for use. The next step is to put everything together to build an application for updating `AttendeeCounter`

The `UpdateOperation` application (Fig. 23.16) demonstrates updating an `Entry` within a transaction. The `main` method constructs an `UpdateInputWindow` to allow the user to select a day and enter a count for the `AttendeeCounter` to update. Method `getServices` (lines 40–50) obtains a reference to the JavaSpaces service.

Method `updateEntry` (lines 53–136) updates an `AttendeeCounter` in the context of a transaction. Lines 67–68 create a transaction by passing the transaction manager and the lease duration to `TransactionFactory` method `create`, which returns a `Transaction.Created` object. Lines 69–70 specify the lease time for the transaction. Line 85 creates an `AttendeeCounter` template that will match all `AttendeeCounters` in the JavaSpaces service. Lines 86–87 take an `Entry` from the JavaSpaces service using `transactionCreated.transaction`. If `UpdateOperation` retrieves an `AttendeeCounter` successfully, then it modifies the `AttendeeCounter` with the day and count information (lines 98–102) and writes it back to the JavaSpaces service. If `UpdateOperation` finds no `AttendeeCounter`, then it does nothing. Lines 115–116 commit the transaction, thus finalizing the transaction and ending the transaction’s lease. If an exception occurs at any time, lines 126–127 abort the transaction and end the transaction’s lease.
// UpdateOperation.java
// This application removes an Entry from the JavaSpace,
// changes the variable's value in the returned Entry and
// deposits the updated Entry into the JavaSpace. All these
// operations are occurred within a transaction.
package com.deitel.advjhtp1.javaspace.update;

// Jini core package
import net.jini.core.lease.Lease;
import net.jini.core.transaction.*;
import net.jini.core.entry.UnusableEntryException;
import net.jini.core.transaction.server.TransactionManager;

// Jini extension package
import net.jini.space.JavaSpace;
import net.jini.lease.*;

// Java core packages
import java.rmi.RemoteException;

// Java extension swing packages
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.javaspace.common.*;

public class UpdateOperation {
    private JavaSpace space;
    private TransactionManager transactionManager;
    private static String hostname = "";
    private static String day = "";
    private static int inputCount = 0;
    private static String output = "\n";

    // default constructor
    public UpdateOperation() {} 

    // constructor gets JavaSpace and TransactionManager
    public void getServices( String jiniURL )
    {
        // get JavaSpace and TransactionManager
        JavaSpaceFinder findtool =
            new JavaSpaceFinder( jiniURL );
        space = findtool.getJavaSpace();
        TransactionManagerFinder findTransaction =
            new TransactionManagerFinder( jiniURL );
        transactionManager =
            findTransaction.getTransactionManager();
    }

Fig. 23.16 Updating an entry using Jini TransactionManager (part 1 of 4).
// update Entry
public void updateEntry( String inputDay, int countNumber )
{
    day = inputDay;
    inputCount = countNumber;

    AttendeeCounter resultCounter = null;
    Transaction.Created transactionCreated = null;
    LeaseRenewalManager manager = new LeaseRenewalManager();

    int oldCount = 0;
    int newCount = 0;

    // create transaction and renew transaction's lease
    try {
        transactionCreated = TransactionFactory.create(
            transactionManager, Lease.FOREVER );
        manager.renewUntil(
            transactionCreated.lease, Lease.FOREVER, null );
    }

    // handle exception creating transaction and renewing lease
    catch ( Exception exception ) {
        exception.printStackTrace();
    }

    // specify matching template, remove template
    // from JavaSpace in transaction, change
    // variable's value and write updated template back
    // to JavaSpace within a transaction
    try {

        // take Entry away from JavaSpace
        AttendeeCounter count = new AttendeeCounter( day );
        resultCounter = ( AttendeeCounter ) space.take( count,
            transactionCreated.transaction, JavaSpace.NO_WAIT );

        // if no matching entry
        if ( resultCounter == null ) {
            // set output message
            output += " No matching Entry is available!\n";
        }
        else { // if find a matching entry

            // update value
            oldCount = resultCounter.counter.intValue();
            newCount = oldCount + inputCount;

            // put updated Entry back to JavaSpace
            resultCounter.counter = new Integer( newCount );
            space.write( resultCounter,
                transactionCreated.transaction, Lease.FOREVER );
        }
    }
}
// output result if transaction completes
output += "Count Information:\n";
output += "   Day: " + resultCounter.day + "\n";
output += "   Old Count: " + oldCount + "\n";
output += "   New Count: " + newCount + "\n";
}

// commit transaction and release lease
transactionCreated.transaction.commit();
manager.remove( transactionCreated.lease );

} // end try

// handle exception updating Entry
catch ( Exception exception ) {
   exception.printStackTrace();

   // revert change and release lease
   try {
      transactionCreated.transaction.abort();
      manager.remove( transactionCreated.lease );
   }

   // handle exception reverting change
   catch ( Exception abortException ) {
      abortException.printStackTrace();
   }
}

} // end method updateEntry

// show output
public void showOutput()
{
   JTextArea outputArea = new JTextArea();
   outputArea.setText( output );
   JOptionPane.showMessageDialog( null, outputArea,
   "UpdateOperation Output", JOptionPane.INFORMATION_MESSAGE );
}

// terminate program
System.exit( 0 );

public static void main( String args[])
{
   // get hostname
   if ( args.length != 1 ) {
      System.out.println("Usage: UpdateOperation hostname");

   Fig. 23.16 Updating an entry using Jini TransactionManager (part 3 of 4).
Figure 23.17 and Fig. 23.18 show the results of running the update `Entry` application. The `WriteOperation Output` window in Fig. 23.17 is the result of running the `WriteOperation` application. We initialize the `Entry` for Wednesday. The `UpdateInputWindow` in Fig. 23.17 is the interface for the user to provide the update information, such as the number of attendees for a seminar on a given day. In that window, we want to send 15 people to attend Wednesday’s seminar. The `UpdateOperation Output` window in Fig. 23.18 shows the result of running the `UpdateOperation` application. The `ReadOperation Output` window in Fig. 23.18 shows the result of reading Wednesday’s `Entry` via the `ReadOperation` application.

To execute this application a Web server, the RMI activation daemon, the Jini lookup service, the JavaSpaces service and the `TransactionManager` service should be running. The following steps compile and execute the `UpdateOperation` application. Ensure that your `CLASSPATH` includes `jini-core.jar`, `jini-ext.jar` and `sun-util.jar`. Compile the java files in the `com\deitel\advjhtp1\javaspace\common` and `com\deitel\advjhtp1\javaspace\update` directories. Run application `UpdateOperation` by specifying the hostname of the Jini lookup service. Do not forget to specify a policy file with the proper permissions.

```java
157         System.exit( 1 );
158     }
159     else
160     hostname = args[ 0 ];
161
162     // get user input day
163     UpdateInputWindow input = new UpdateInputWindow( hostname );
164
165     } // end method main
166 }
```

**Fig. 23.17** WriteOperation Output and UpdateInputWindow user interface.
23.12 Case Study: Distributed Image Processing

Image processing can be a time-consuming task, especially for large images. In this case study, we use JavaSpaces services to build a distributed image-processing system for applying filters to images (e.g., blur, sharpen, etc.). We define class `ImageProcessorClient` to partition a large image into smaller pieces and write these pieces into a JavaSpaces service. Multiple `ImageProcessor` run in parallel to process the smaller images by applying appropriate filters, then write the processed images back into the JavaSpaces service. The `ImageProcessorClient` then takes the processed subimages from the JavaSpaces service and builds the complete, processed image. Figure 23.19 shows the basic structure of the `ImageProcessor` application.
23.12.1 Defining an Image Processor

In this case study, the distributed infrastructure consists of a set of dedicated image-processor nodes that retrieve Entries from a JavaSpaces service. Each of these nodes processes an Entry and writes it back to the JavaSpaces service. The image processors have four filters—blur, color, invert and sharpen. A blur filter blurs an image. A color filter alters the RGB color bands of an image. An invert filter inverts an image’s RGB color values. A sharpen filter sharpens the edges of an image. Each image-processor node in this distributed system polls the JavaSpaces service constantly looking for Entries to process.

Applications can use the distributed system to process images. The application in our case study is ImageProcessorClient (Fig. 23.23). ImageProcessorClient will prompt the user for a file name and the number of pieces in which to divide the original image. ImageProcessorClient then writes the pieces to the JavaSpaces service. ImageProcessorClient will then poll the JavaSpaces service until it retrieves all of the processed pieces. Finally, the application will assemble the image and display the results to the user.

Class ImageEntry (Fig. 23.20) defines the Entries that the application can store in the JavaSpaces service. Lines 16–20 define the public fields of the Entry. Line 23 defines the required empty constructor. Lines 26–34 define a constructor that initializes all fields in ImageEntry. Lines 37–41 define a constructor that initializes the name and processed fields. Lines 44–47 define a constructor that initializes the name field.

Class ImageProcessor (Fig. 23.21) represents each node of the image-processing distributed system that is capable of processing images. Each ImageProcessor node polls the JavaSpaces service for ImageEntries. When a client writes an unprocessed ImageEntry into the JavaSpaces service, the first ImageProcessor node to retrieve that Entry will process it. Each ImageProcessor node creates a transaction for each Entry it retrieves and doesn’t commit the transaction until it successfully writes the processed ImageEntry back to the JavaSpaces service, so if an ImageProcessor were to fail, the ImageEntry would not be lost.

```java
// Fig. 23.20 ImageEntry.java
// This class defines the Entry for the image.
package com.deitel.advjhtp1.javaspace.ImageProcessor;

// Java core packages
import java.util.*;

// Java standard extension
import javax.swing.ImageIcon;

// Jini core packages
import net.jini.core.entry.Entry;

public class ImageEntry implements Entry {
    public String name;
    public String filter;
}
```

Fig. 23.20 ImageEntry defines the Entries to store in the JavaSpaces service (part 1 of 2).
```java
public Integer number;
public Boolean processed;
public ImageIcon imageIcon;

// empty constructor
public ImageEntry() {}

// ImageEntry constructor
public ImageEntry( String imageName, String imageFilter,
                    int order, boolean done, ImageIcon icon )
{
    name = imageName;
    filter = imageFilter;
    number = new Integer( order );
    processed = new Boolean( done );
    imageIcon = icon;
}

// ImageEntry constructor
public ImageEntry( String imageName, boolean done )
{
    name = imageName;
    processed = new Boolean( done );
}

// ImageEntry constructor
public ImageEntry( String imageName )
{
    name = imageName;
}
```

**Fig. 23.20** ImageEntry defines the Entries to store in the JavaSpaces service (part 2 of 2).

```java
// Fig. 23.21 ImageProcessor.java
// Takes entries from the JavaSpace, applies a filter to
// the image piece, and writes processed entry back to JavaSpace.
package com.deitel.advjhtp1.javaspace.ImageProcessor;

// Java standard extensions
import javax.swing.*;

// Jini core packages
import net.jini.core.lease.Lease;
import net.jini.core.transaction.*;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.entry.*;
import net.jini.core.transaction.*;
import net.jini.lease.*;
```

**Fig. 23.21** Image processing node that uses the JavaSpaces service (part 1 of 4).
import net.jini.space.JavaSpace;

import com.deitel.advjhtp1.javaspace.common.*;

public class ImageProcessor {

    private JavaSpace space;
    private TransactionManager manager;

    // ImageProcessor constructor
    public ImageProcessor ( String hostname )
    {
        // get the JavaSpace
        String jiniURL = "jini://" + hostname;
        JavaSpaceFinder finder =
            new JavaSpaceFinder( jiniURL );
        space = finder.getJavaSpace();

        // get the TransactionManager
        TransactionManagerFinder findTransaction =
            new TransactionManagerFinder( jiniURL );
        manager =
            findTransaction.getTransactionManager();
    }

    // wait for unprocessed image
    public void waitForImage()
    {
        LeaseRenewalManager leaseManager =
            new LeaseRenewalManager();
        while ( true ) {
            try {
                Transaction.Created transactionCreated =
                    TransactionFactory.create(
                        manager, Lease.FOREVER );

                // renew transaction's lease
                leaseManager.renewUntil(
                    transactionCreated.lease, Lease.FOREVER, null );

                ImageEntry template = new ImageEntry( null, false );
                ImageEntry entry = ( ImageEntry ) space.take(
                    template, transactionCreated.transaction,
                    Lease.FOREVER );
            }
        }
    }
}
if ( entry != null ) {

    // get image icon
    ImageIcon imageIcon = entry.imageIcon;

    Filters filters = new Filters( imageIcon );

    if ( entry.filter.equals( "BLUR" ) )
        filters.blurImage();

    else if ( entry.filter.equals( "COLOR" ) )
        filters.colorFilter();

    else if ( entry.filter.equals( "INVERT" ) )
        filters.invertImage();

    else if ( entry.filter.equals( "SHARP" ) )
        filters.sharpenImage();

    // update the fields of result entry
    entry.imageIcon = filters.getImageIcon();
    entry.processed = new Boolean( true );

    // put the updated Entry back to JavaSpace
    Lease writeLease = space.write( entry, transactionCreated.transaction, Lease.FOREVER );
    leaseManager.renewUntil( writeLease, Lease.FOREVER, null );

} // end if

// commit the transaction and release the lease
transactionCreated.transaction.commit();
leaseManager.remove( transactionCreated.lease );

} // end try

// handle exception
catch ( Exception exception ) {
    exception.printStackTrace();
}

} // end while

} // end method wait for images

public static void main( String[] args )
{

    // get the hostname
    if ( args.length != 1 ) {
        System.out.println(
            "Usage: ImageProcessor hostname" );
    }
Constructor `ImageProcessor` (lines 29–42) gets a JavaSpaces service and a TransactionManager service from the user-specified hostname. Classes `JavaSpaceFinder` and `TransactionManagerFinder` are defined in Fig. 23.1 and Fig. 23.15 respectively. Method `waitForImage` (lines 45–112) waits for an unprocessed image. Lines 50–110 cycle indefinitely to process `Entry`s from the JavaSpaces service. Line 62 defines the `Entry` template that `ImageProcessor` will use to retrieve unprocessed `ImageEntry` from the JavaSpaces service. Lines 63–65 retrieve an object from the JavaSpaces service. If the `Entry` retrieved from the JavaSpaces service is not `null`, lines 70–84 apply a filter to the image. Lines 87–88 set fields in the `ImageEntry` with values that will identify the `ImageEntry` as a processed `Entry`. Lines 91–91-93 write the processed `ImageEntry` back to the JavaSpaces service. Line 100 commits the transaction. Line 101 expires the lease on the transaction.

Method `main` (lines 114–129) gets the user-specified hostname and calls `ImageProcessor` constructor. Method `waitForImage` starts polling the JavaSpaces service for an unprocessed image.

Class `Filters` (Fig. 23.22) filters an image. It provides four types of filters—blur, sharpen, invert and color. These four filters can be found in the Chapter 4 package `com.deitel.advjhtp.java2d`. The `Filters` constructor (lines 26–40) takes an `ImageIcon` and converts the `ImageIcon` to the `BufferedImage`. Method `blurImage` (lines 43–46) applies a blur filter to the `BufferedImage`. Method `sharpenImage` (lines 49–53) applies a sharpen filter to the `BufferedImage`. Method `invertImage` (lines 56–60) applies an invert filter to the `BufferedImage`. Method `colorFilter` (lines 63–67) applies a color filter to a `BufferedImage`. Method `getImageIcon` (lines 70–73) returns the filtered image as an `ImageIcon`.

```
120  System.exit( 1 );
121 }
122
123  ImageProcessor processor =
124      new ImageProcessor( args[ 0 ] );
125
126  // wait for image
127  processor.waitForImage();
128
129  } // end method main
130 }
```

**Fig. 23.21** Image processing node that uses the JavaSpaces service (part 4 of 4).

**Fig. 23.22** Class `Filters` applies a Java 2D filter to an image (part 1 of 3).
// Java standard extensions
import javax.swing.*;

// Deitel packages
import com.deitel.advjhtp1.java2d.*;

public class Filters {
    Java2DImageFilter blurFilter;
    Java2DImageFilter sharpenFilter;
    Java2DImageFilter invertFilter;
    Java2DImageFilter colorFilter;

    BufferedImage bufferedImage;

    // constructor method initializes ImageFilters and pulls
    // BufferedImage out of ImageIcon
    public Filters( ImageIcon icon ) {
        blurFilter = new BlurFilter();
        sharpenFilter = new SharpenFilter();
        invertFilter = new InvertFilter();
        colorFilter = new ColorFilter();

        Image image = icon.getImage();
        bufferedImage = new BufferedImage(
            image.getWidth( null ), image.getHeight( null ),
            BufferedImage.TYPE_INT_RGB );

        Graphics2D gg = bufferedImage.createGraphics();
        gg.drawImage( image, null, null );
    }

    // apply BlurFilter to bufferedImage
    public void blurImage() {
        bufferedImage = blurFilter.processImage( bufferedImage );
    }

    // apply SharpenFilter to bufferedImage
    public void sharpenImage() {
        bufferedImage = sharpenFilter.processImage( bufferedImage );
    }

    // apply InvertFilter to bufferedImage
    public void invertImage() {
        bufferedImage = invertFilter.processImage( bufferedImage );
    }
}

Fig. 23.22 Class Filters applies a Java 2D filter to an image (part 2 of 3).
23.12.2 Partitioning an Image into Smaller Pieces

To parallelize the task of filtering a large image, we can partition the image into smaller pieces and let multiple ImageProcessors concurrently filter the smaller images. Then, we can combine the processed pieces to create the complete, processed image. Class ImageProcessorClient (Fig. 23.23) prompts the user for an image file to process and partitions the large image. ImageProcessorClient wraps the image pieces in ImageEntrys and write them into the JavaSpaces service. Once ImageProcessorClient writes all image pieces into the JavaSpaces service, it reads back all processed ImageEntrys.

The ImageProcessorClient constructor (lines 42–192) defines the user interface to get the user input data. Lines 66–81 display a JChooserPanel when a user presses the Choose file to process button to prompt the user for an image file. Lines 90–111 define a JLabel and a JTextField which obtains the user-specified number of subimages. Lines 114–135 create a label and a drop-down box that allows the user to specify the filter to apply on the image. Lines 142–186 add an OK button to the window. Lines 152–186 define the action when the user clicks the OK button. The application first checks whether the user specified the image name and partition number. If so, the application passes the user-specified data to the ImageSeparator (Fig. 23.24) constructor and calls methods partitionImage and storeImage of class ImageSeparator to partition the image and store the sub images in a JavaSpaces service, respectively. Method collect obtains a reference to the JavaSpaces service (lines 198–201). Lines 211–212 verify that the user input number divides evenly between columns and rows. If the user-input number does not divide evenly, then the image will be broken into 4 pieces. Line 215 creates a template that will match all processed image pieces. Line 218 creates a snapshot of the template. Lines 221–225 retrieve all processed ImageEntrys from the JavaSpaces service. Lines 251–265 reassemble the image and display the resulting image in a separate window. Lines 267–274 display an error message if the file was not found. Method main (lines 278–294) gets the user-specified hostname from the command line and displays the GUI.

Class ImageSeparator (Fig. 23.24) partitions an image into smaller pieces and places the subimages into a JavaSpaces service. The constructor (lines 33–39) gets the image name, number of subimages and this filter to be apply to the image. Method partitionImage (lines 42–50) gets the ImageIcon from the image file (lines 44–45) and calls method parseImage of class ImageParser (Fig. 23.25) to partition the image.
into smaller pieces. Method `displayImage` (lines 53–60) creates a `ImageDisplayer` (Fig. 23.26) to display the original image. Method `storeImage` (lines 63–101) writes the subimages into a JavaSpaces service. Lines 77–88 write each image into the JavaSpaces service and renew the image’s lease to `Lease.FOREVER`.

```java
// Fig. 23.23 ImageProcessingClient.java
// The application asks for user-specified image file, image
// filter type and image partition number, filters the image
// and displays the processed image.
package com.deitel.advjhtpl.javaspace.ImageProcessor;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

// Java extension packages
import javax.swing.*;

// Jini core packages
import net.jini.core.lease.Lease;
import net.jini.core.entry.*;
import net.jini.core.transaction.*;
import net.jini.core.transaction.server.TransactionManager;

// Jini extension packages
import net.jini.space.JavaSpace;

// Deitel packages
import com.deitel.advjhtpl.javaspace.common.*;

public class ImageProcessingClient extends JFrame {

    private String[] operations = { "BLUR", "COLOR", "INVERT", "SHARP" };
    private JButton okButton;
    private JComboBox operationComboBox;
    private JTextField imageText;
    private JTextField numberText;
    private static String hostname = "";
    private String imageName;
    private String operation = "BLUR";
    private int partitionNumber = 0;

    public ImageProcessingClient( String host ) {
        super ( "ImageProcessInput" );
        hostname = host;
        Container container = getContentPane();
```
// define the center panel
JPanel centerPanel = new JPanel();
centerPanel.setLayout( new GridLayout( 3, 2, 0, 5 ) );

// add image label
JLabel imageLabel = new JLabel( "Image File:",
    SwingConstants.CENTER );
centerPanel.add( imageLabel );

Button openFile = new JButton( "Choose file to process" );
openFile.addActionListener(
    new ActionListener() {

        public void actionPerformed( ActionEvent event ) {
            JFileChooser fileChooser = new JFileChooser();
            fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
            int result = fileChooser.showOpenDialog( null );
            File file;

            // user clicked Cancel button on dialog
            if ( result == JFileChooser.CANCEL_OPTION )
                file = null;
            else {
                file = fileChooser.getSelectedFile();
                imageName = file.getPath();
            }
        } // end method actionPerformed
    } // end ActionListener constructor
); // end addActionListener

centerPanel.add( openFile );

// add number label
JLabel numberLabel = new JLabel( "Partition Number:",
    SwingConstants.CENTER );
centerPanel.add( numberLabel );

// add number text field
numberText = new JTextField( 10 );
centerPanel.add( numberText );

// install a listener to the number text field
numberText.addActionListener(  

Fig. 23.23 Image-processing distributed system client (part 2 of 6).
new ActionListener() {
    // get the text when the user feeds a return
    // character in the text field
    public void actionPerformed( ActionEvent event ) {
        partitionNumber = Integer.parseInt(
            event.getActionCommand() );
    }
};

// add operation label
JLabel operationLabel = new JLabel( "Operation Type:",
    SwingConstants.CENTER );
centerPanel.add( operationLabel );

// add a combo box
operationComboBox = new JComboBox( operations );
operationComboBox.setSelectedIndex( 0 );
centerPanel.add( operationComboBox );

// install a listener to the combo box
operationComboBox.addItemListener(
    new ItemListener() {
        // an operation other than BLUR is selected
        public void itemStateChanged( ItemEvent itemEvent ) {
            operation =
                ( String ) operationComboBox.getSelectedItem();
        }
    });

// define the button panel
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout( new GridLayout( 1, 1, 0, 5 ) );

// add the OK button
okButton = new JButton( "OK" );
buttonPanel.add( okButton );

// add a listener to the OK button
okButton.addActionListener(
    new ActionListener() {

public void actionPerformed( ActionEvent event )
{
    // get user inputs
    partitionNumber = Integer.parseInt(
        numberText.getText() );

    // check whether the user
    // fills in both text fields
    if ( ( partitionNumber == 0 )
        || ( imageName == null ) ) {
        JOptionPane.showMessageDialog( null,
            "Either image name or partition number 
            + "is not specified!", "Error",
            JOptionPane.ERROR_MESSAGE);
    }
    else {
        setVisible( false );

        // partition the image into smaller pieces
        // and store the sub images into a JavaSpace
        ImageSeparator imageSeparator =
            new ImageSeparator( 
                imageName, operation, partitionNumber );
        imageSeparator.partitionImage();
        imageSeparator.storeImage( hostname );
        imageSeparator.displayImage();
        collect();
    }
}

} // end method actionPerformed

} // end ActionListener constructor

}; // end addActionListener

// put everything together
container.add( centerPanel, BorderLayout.CENTER );
container.add( buttonPanel, BorderLayout.SOUTH );

} // end ImageProcessingClient constructor

// collect processed images
public void collect()
{
    // get the JavaSpace
    String jiniURL = "jini://" + hostname;
    JavaSpaceFinder findtool =
        new JavaSpaceFinder( jiniURL );
    JavaSpace space = findtool.getJavaSpace();

Fig. 23.23 Image-processing distributed system client (part 4 of 6).
Vector unOrderedImages = new Vector();
Vector orderedImages = null;

// removes all images in the JavaSpace
// that have the specified name
try {
    double squareRoot = Math.sqrt( partitionNumber );

    if ( Math.floor( squareRoot ) != ( squareRoot ) )
        partitionNumber = 4;

    // specify the matching template
    ImageEntry template = new ImageEntry( imageName, true );

    // snapshot the template
    Entry snapshotEntry = space.snapshot( template );

    // collect images
    for ( int i = 0; i < partitionNumber ; i++ ) {
        ImageEntry remove = ( ImageEntry ) space.take(
            snapshotEntry, null, Lease.FOREVER );
        unOrderedImages.add( remove );
    }

    int imageCount = unOrderedImages.size();
    orderedImages =
        new Vector( imageCount );

    // initialize the Vector
    for ( int i = 0; i < imageCount; i++ )
        orderedImages.add( null );

    // order the sub images
    for ( int i = 0; i < imageCount; i++ ) {
        ImageEntry image =
            ( ImageEntry ) unOrderedImages.elementAt( i );
        orderedImages.setElementAt( image.imageIcon, image.number.intValue() );
    }

} // end try

// handle exception collecting images
catch ( Exception exception ) {
    exception.printStackTrace();
}

// put images together and display the result image
if ( orderedImages.size() > 0 ) {
    ImageParser imageParser = new ImageParser();
    ImageIcon icon = imageParser.putTogether( orderedImages );
}
ImageDisplayer imageDisplayer =
    new ImageDisplayer( icon );
imageDisplayer.setSize( icon.getIconWidth() + 50,
    icon.getIconHeight() + 50 );
imageDisplayer.setVisible( true );
imageDisplayer.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE );

else {
    JOptionPane.showMessageDialog( null,
        "Invalid image name", "Error",
        JOptionPane.ERROR_MESSAGE);
    // terminate program
    System.exit( 0 );
}

// end method collect

public static void main( String[] args )
{
    // get the hostname
    if ( args.length != 1 ) {
        System.out.println(
            "Usage: ImageProcessingClient hostname" );
        System.exit( 1 );
    }

    ImageProcessingClient processor =
        new ImageProcessingClient( args[ 0 ] );
    // set the window size and display it
    processor.setSize( 350, 150 );
    processor.setVisible( true );

} // end method main

Fig. 23.23 Image-processing distributed system client (part 6 of 6).

// Fig. 23.24 ImageSeparator.java
// This class partitions the image into smaller pieces evenly
// and stores the smaller images into a JavaSpace.
package com.deitel.advjhtp1.javaspace.ImageProcessor;

// Java core packages
import java.util.*;
import java.rmi.*;

Fig. 23.24 Partitioning an image into smaller pieces and storing subimages in a JavaSpaces service (part 1 of 3).
// Java standard extensions
import javax.swing.*;

// Jini core packages
import net.jini.core.lease.Lease;
import net.jini.core.transaction.TransactionException;

// Jini extension packages
import net.jini.space.JavaSpace;
import net.jini.lease.*;

// Deitel packages
import com.deitel.advjhtp1.javaspace.common.*;

public class ImageSeparator {
    private String imageName;
    private String filterType;
    private int partitionNumber;
    private Vector imagePieces;
    private ImageIcon icon;

    // ImageSeparator constructor
    public ImageSeparator(
        String name, String type, int number )
    {
        imageName = name;
        filterType = type;
        partitionNumber = number;
    }

    // partition the image into smaller pieces evenly
    public void partitionImage() {
        ImageParser imageParser = new ImageParser();
        icon = new ImageIcon( imageName );

        // partition the image
        imagePieces = imageParser.parseImage( icon, partitionNumber );
    }

    // display the image
    public void displayImage( ) {
        ImageDisplayer imageDisplayer =
            new ImageDisplayer( icon );
        imageDisplayer.setSize( icon.getIconWidth() + 50,
                               icon.getIconHeight() + 50 );
        imageDisplayer.setVisible( true );
    }
}

Fig. 23.24 Partitioning an image into smaller pieces and storing subimages in a JavaSpaces service (part 2 of 3).
Class **ImageParser** (Fig. 23.25) partitions an image into smaller pieces and puts sub-images together to reform a large image. Method **parseImage** (lines 25–66) partitions an image into smaller pieces. If the number of subimages specified by the user does not divide evenly between the number of rows and columns, then the default number 4 is used (lines 29–35). Lines 42–62 partition the image. The resulting subimages are stored in a **Vector**. Method **putTogether** (lines 71–106) puts subimages back together. Lines 93–102 get the subimages from the input **Vector** and piece them together into a **BufferedImage** to form a complete image. Line 104 returns the image as an **ImageIcon**.
Class **ImageDisplayer** (Fig. 23.26) is a **JFrame** subclass for displaying an image. The constructor (lines 15–30) gets an **ImageIcon** and displays it in a **JLabel**.

```java
// Fig. 23.25: ImageParser.java
// This class partitions an image into smaller pieces.
package com.deitel.advjhtp1.javaspace.ImageProcessor;

// Java core packages
import java.awt.image.*;
import java.net.URL;
import java.awt.*;
import java.lang.*;
import java.util.Vector;
import java.awt.geom.*;

// Java standard extensions
import javax.swing.*;

public class ImageParser {

    ImageIcon image;

    public ImageParser() {} // pass parseImage an ImageIcon on the number of piece
    // you want it split into the number of piece must be a
    // perfect square - this can be extended later
    public Vector parseImage(ImageIcon imageIcon, int numberPieces) {
        Vector vector = new Vector();
        double squareRoot = Math.sqrt(numberPieces);
        if (Math.floor(squareRoot) != squareRoot) {
            System.out.println("This is not a square number, " + " setting to default...");
            numberPieces = 4;
        }
        int numberRows = (int) Math.sqrt(numberPieces);
        int numberColumns = (int) Math.sqrt(numberPieces);
        // retrieve Image from BufferedImage
        Image image = imageIcon.getImage();
        BufferedImage bufferedImage = new BufferedImage(image.getWidth(null), image.getHeight(null), BufferedImage.TYPE_INT_RGB);
        Graphics2D g2D = bufferedImage.createGraphics();
        g2D.drawImage(image, null, null);
        // get number of rows and columns
        // retrieve Image from BufferedImage
        Image image = imageIcon.getImage();
        BufferedImage bufferedImage = new BufferedImage(image.getWidth(null), image.getHeight(null), BufferedImage.TYPE_INT_RGB);
        Graphics2D g2D = bufferedImage.createGraphics();
        g2D.drawImage(image, null, null);
    }
}
```

**Fig. 23.25** Partitioning and reforming an image (part 1 of 3).
// get size of each piece
int width = bufferedImage.getWidth() / numberColumns;
int height = bufferedImage.getHeight() / numberRows;

// make each of images
for (int x = 0; x < numberRows; x++) {
    for (int y = 0; y < numberColumns; y++) {
        vector.add(new ImageIcon(
            bufferedImage.getSubimage( x * width, y * height, width, height )
        ));
    }
}

return vector;

} // end method parseImage

// takes a vector of image icons (must be a square number)
// of elements and returns an image icon with the images
// put back together again
public ImageIcon putTogether( Vector vector ) {
    double size = vector.size();
    int numberRowColumn = (int) Math.sqrt(size);

    // step 1, get first Image
    Image tempImage =
        ((ImageIcon) vector.get(0)).getImage();

    // get total size of one piece
    int width = tempImage.getWidth(null);
    int height = tempImage.getHeight(null);

    // create buffered image
    BufferedImage totalPicture = new BufferedImage(
        width * numberRowColumn, height * numberRowColumn,
        BufferedImage.TYPE_INT_RGB
    );

    // create Graphics for BufferedImage
    Graphics2D graphics = totalPicture.createGraphics();

    // draw images from the vector into buffered image
    for (int x = 0; x < numberRowColumn; x++) {
        for (int y = 0; y < numberRowColumn; y++) {
            Image image = ((ImageIcon) vector.get( y + numberRowColumn * x )).getImage();
            graphics.drawImage(image,
                AffineTransform.getTranslateInstance( x * width, y * height ), null);
        }
    }
}

Fig. 23.25 Partitioning and reforming an image (part 2 of 3).
23.12.3 Compiling and Running the Example

Before compiling the sample code, ensure that `jni-core.jar`, `jni-ext.jar`, and `sun-util.jar` are included in the `CLASSPATH`. To execute the example, start:

1. a Web server,
2. the RMI activation daemon,
3. the Jini lookup service,
4. the JavaSpaces service, and
5. the Jini `Transaction` service.
On a second computer, perform the following tasks:

1. compile all java files in directory com\deitel\advjhtpl\java-space\ImageProcessor, and

2. start the **ImageProcessorClient** application by specifying the hostname of the computer running the Jini lookup service (you also must specify an appropriate policy file).

On at least one other machine, start an **ImageProcessor** node and specify the hostname of the computer running the Jini lookup service (you also must specify an appropriate policy file).

One important thing to note in this example is that doubling the number of **ImageProcessor** nodes that retrieve **Entry**s from the JavaSpaces service doubles the processing power of the application. If you do not have multiple computers, you can run the entire application on a single computer. Although this defeats the purpose of a distributed system, running the application on a single computer can be useful for testing and debugging the application. Figure 23.27 displays the GUI for the **ImageProcessorClient** application.

Fig. 23.28 displays the image on the left-hand side before application **ImageProcessorClient** distributes the image to the **ImageProcessor** nodes. The **ImageProcessor** nodes apply the blur filter to the image. The image on the right-hand side shows the resulting image after **ImageProcessorClient** assembles the processed pieces.

---

**Fig. 23.27** GUI from **ImageProcessorMain** and **ImageCollector** applications.

**Fig. 23.28** Images before and after blurring.
23.13 Internet and World Wide Web Resources

www.sun.com/jini/specs/js1_1.pdf
This site provides the Jini JavaSpaces service Specification.

These JavaWorld online magazine URLs represent a series of articles that introduce JavaSpaces technology, including the available operations, transactions and leasing.

www.byte.com/documents/s=146/BYT19990921S0001/index.htm
This article provides an overview of JavaSpaces technology and introduces its basic concepts.

SUMMARY

• A JavaSpaces service is a Jini service that implements a simple, high-level architecture for building distributed systems. The JavaSpaces service enables Java objects to communicate, share objects and coordinate tasks using an area of shared memory.

• A JavaSpaces service provides three fundamental operations—write, take and read. The write operation places an object—called an entry—into the JavaSpaces service.

• The take operation specifies a template and removes from the JavaSpaces service an entry that matches the given template. The read operation is similar to the take operation, but does not remove the matching entry from the JavaSpaces service.

• In addition to the three basic operations, JavaSpaces services support transactions through the Jini transaction manager, and a notification mechanism that notifies an object when an entry that matches a given template is written to the JavaSpaces service.

• An entry stored in a JavaSpaces service will remain in the JavaSpaces service until its lease expires or until a program takes the entry from the JavaSpaces service.

• A JavaSpaces service locates objects by comparing those objects to a template. The template specifies the search criteria against which the JavaSpaces service compares each entry. When one or more entries match the template, the JavaSpaces service returns a single matching entry.

• JavaSpaces services can use the Jini transaction manager to ensure operations execute atomically.

• Objects in a JavaSpaces service are shared. Programs can read and take entries from the JavaSpaces service, modify the public fields in those entries and write them back to the JavaSpaces service for other programs to use.

• Any object stored in the JavaSpaces service must implement interface Entry (package net.jini.core.entry). JavaSpaces services Entries adhere to the Jini Entry contract defined in the Jini Core Specification.

• An Entry can have multiple constructors and as many methods as required. Other requirements include a public no-argument constructor, public fields and no-primitive type fields. The JavaSpaces service proxy uses the no-argument constructor to instantiate the matching Entry during the deserialization process.

• All fields that will be used as the template matching fields in an Entry must be public (for more information on template matching fields, see Section 23.8). As defined by the Jini Core Specification, an Entry cannot have primitive-type fields.

• JavaSpaces technology, like Jini, requires several underlying services. The JavaSpaces service depends on the Jini lookup service. When transactions are required, the Jini transaction service must be started. JavaSpaces services also depend on a Web server and rmid
Two versions of JavaSpaces services are available. One is the transient JavaSpaces service (non-activatable). The other is the persistent JavaSpaces service (activatable).

The transient JavaSpaces service does not require the RMI activation daemon (`rmid`), because the transient JavaSpaces service is not activatable. Once the transient JavaSpaces service terminates, all state information is lost and `rmid` is unable to restart the service.

The persistent JavaSpaces service is activatable, and therefore requires the RMI activation daemon. If the persistent JavaSpaces service terminates, all of its state information is stored in a log file and `rmid` can restart the service at a later time.

Upon initialization, each JavaSpaces service registers itself with local Jini lookup services.

Clients access objects in a JavaSpaces service through interface `JavaSpace` (package `net.jini.space`). Interface `JavaSpace` provides several methods—`notify`, `read`, `readIfExists`, `take`, `takeIfExists`, `write` and `snapshot`.

The `write` operation inserts an `Entry` into a JavaSpaces service. If an identical `Entry` already exists in the JavaSpaces service, this operation does not overwrite the existing `Entry`. Instead, the `write` operation places a copy of the `Entry` into the JavaSpaces service.

The `read` operation attempts to read an `Entry` that matches an `Entry` template from a JavaSpaces service. If no matching `Entry` exists in the JavaSpaces service, this operation does not return an `Entry`. Instead, the `read` operation places a copy of the `Entry` into the JavaSpaces service.

Method `readIfExists` checks to see if a matching `Entry` exists within the JavaSpaces service. If an `Entry` does not exist in the JavaSpaces service, method `readIfExists` should return `null` immediately. Method `readIfExists` does not block unless the matching `Entry` is a participant in an uncommitted transaction.

The `take` operation attempts to remove an `Entry` that matches an `Entry` template from a JavaSpaces service. This operation works like the `read` operation, except that the `take` operation removes the matching `Entry` from the JavaSpaces service.

Method `take` blocks until a matching `Entry` is found in the JavaSpaces service or until a time-out occurs.

Method `takeIfExists` checks to see if a matching `Entry` exists within the JavaSpaces service. If an `Entry` does not exist in the JavaSpaces service, method `takeIfExists` should return `null` immediately. Method `takeIfExists` does not block unless the matching `Entry` is part of an uncommitted transaction.

The `notify` operation requests that the JavaSpaces service sends a notification to a listener object when a client writes a matching `Entry` into the JavaSpaces service. With this method, an application does not need to check repeatedly for a `Entry` in a JavaSpaces service.

Method `snapshot` increases performance when a program must serialize one `Entry` repeatedly. Each time a program transfers an `Entry` into a JavaSpaces service (e.g., by writing that `Entry` or by using that `Entry` as a template), that `Entry` must be serialized. Invoking method `snapshot` serializes the `Entry` once and reuses this serialized `Entry` for future transfers.

Method `write` takes three arguments—an `Entry`, a `Transaction` object and a `long` value that requests an amount of time for which the JavaSpaces service should keep the `Entry`. The `long` value represents the lease length for the `Entry`.

The `read` and the `take` operations retrieve `Entries` from a JavaSpaces service. A client can read or take an `Entry` from the JavaSpaces service by supplying a template `Entry` against which to compare the public fields of `Entries` in the JavaSpaces service. The template indicates which fields should be used for comparison purposes.
• The retrieval process is based on a template-matching mechanism, which matches *Entry*s according to the values of their *public* fields. Fields in the template with *non-null* values must match with their *Entry* counterparts in the JavaSpaces service exactly. Fields in the template that are set to *null* act as wildcards.

• If a set of *Entry*s of the same type exist within a JavaSpaces service, only those fields which equal those of the template are used to match an *Entry* or a set of *Entry*s contained in the JavaSpaces service. Fields in the template set to *null* can have their matching counterparts in the JavaSpaces service have any value in the corresponding field(s).

• The *read* operation obtains *Entry*s without removing them from the JavaSpaces service. Methods *read* and *readIfExists* perform the read operation. Each method takes three arguments—an *Entry* that specifies the template to match, a *Transaction* object and a *long* value.

• Method *read* specifies a period of time for which the read operation should block before simply returning *null*. Method *readIfExists* is a non-blocking version of method *read*. If there are no matching *Entry*s, *readIfExists* returns *null* immediately.

• Method *readIfExists* blocks only if the developer specifies a period of time for which the *readIfExists* waits if, at first, the matching *Entry* is a part of an incomplete transaction. If the matching *Entry* is not involved in any transaction, then the *read* operation returns the matching *Entry* immediately.

• Both the *read* and *readIfExists* methods return only one *Entry*. If multiple *Entry*s match the template, the *read* operation picks one arbitrarily.

• The *take* operation obtains an *Entry* and removes it from the JavaSpaces service. Methods *take* and *takeIfExists* perform the take operation. Methods *take* and *takeIfExists* are similar to methods *read* and *readIfExists*. The only difference is that the matching *Entry* returned by a *take* or *takeIfExists* operation is removed from the JavaSpaces service.

• The *notify* operation asks the JavaSpaces service to send a notification to a listener when a client writes a matching *Entry* into the JavaSpaces service. Method *notify* takes five parameters—an *Entry* that specifies the matching template, a *Transaction* object, a listener that implements interface *RemoteEventListener* (package *net.jini.core.event*), a *long* value that specifies the lease time for the registration of the listener and a *MarshalledObject* (package *java.rmi*) that the JavaSpaces service will pass to the remote listener as part of a notification.

• Method *snapshot* takes a matching template and returns a specialized representation of the *Entry* (a snapshot *Entry*). This snapshot *Entry* can be used only in the JavaSpaces service that generated it.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Term</th>
<th>Class/Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort method</td>
<td>Transaction</td>
</tr>
<tr>
<td>commit method</td>
<td>Transaction</td>
</tr>
<tr>
<td><em>Entry</em> class (net.jini.core.entry)</td>
<td></td>
</tr>
<tr>
<td>JavaSpaces interface (net.jini.space)</td>
<td>JavaSpaces service</td>
</tr>
<tr>
<td>notify method</td>
<td>JavaSpace</td>
</tr>
<tr>
<td>outrigger JavaSpace implementation</td>
<td>persistent JavaSpaces service</td>
</tr>
<tr>
<td>read method</td>
<td>JavaSpace</td>
</tr>
<tr>
<td>readIfExists method</td>
<td>template matching mechanism</td>
</tr>
<tr>
<td>Transaction.Created class</td>
<td>Transaction.Created.transaction</td>
</tr>
<tr>
<td><em>RemoteEventListener</em> interface</td>
<td>TransactionException class</td>
</tr>
<tr>
<td><em>UnusableEntryException</em> class</td>
<td>TransactionManager service</td>
</tr>
<tr>
<td><em>UnusableEntryException</em> class</td>
<td>transient JavaSpaces service</td>
</tr>
<tr>
<td><em>write</em> method</td>
<td>JavaSpace</td>
</tr>
</tbody>
</table>


SELF-REVIEW EXERCISES

23.1 Fill in the blanks in each of the following statements:
   a) The JavaSpaces technology is an ________ queue.
   b) Objects placed in the JavaSpaces service must implement the ________ interface.
   c) Two methods of the JavaSpace interface that read an Entry from the JavaSpaces service are ________ and ________.
   d) Creating a transaction, requires a ________.
   e) Method ________ can be used to avoid unnecessary serializations of an Entry.

23.2 State whether each of the following is true or false. If false, explain why.
   a) Entries may have private fields that are used for template matching.
   b) Method take can remove all matching Entries from the JavaSpaces service.
   c) To modify an Entry in a JavaSpaces service, we need to take the Entry away from the JavaSpaces service, change the values of the Entry and put the Entry back to the JavaSpaces service.
   d) Objects stored in a JavaSpaces service cannot have primitive type fields.
   e) When writing an Entry to a JavaSpaces service, if the Entry already exists, the new Entry overwrites the old Entry.

ANSWERS TO SELF-REVIEW EXERCISES

23.1 a) object-based message-oriented. b) Entry. c) read, readIfExists. d) transaction manager. e) snapshot.

23.2 a) False. The fields in an Entry used for template matching must be public because the JavaSpaces service needs to access these fields. b) False. If multiple Entries match the template, method take removes only one. c) True. d) True. e) False. JavaSpaces services can have multiple copies of the same Entry. No matter whether an Entry exist, the write operation puts the Entry into the JavaSpaces service, unless an exception occurs during the write process.

EXERCISES

23.3 Describe the difference between method take and method takeIfExists in the JavaSpace interface.

23.4 Start a JavaSpaces service with a name other than the default name and find this JavaSpaces service with Jini lookup service using a NameEntry.

23.5 In the WriteOperation application (Fig. 23.3), the Entries written into the JavaSpaces service are kept in the JavaSpaces service for at most 5 minutes. Using LeaseRenewalManager, rewrite the application so that the written Entries are kept in the JavaSpaces service forever.

23.6 Write a program that uses method take to remove all Entries that match a specific template. Compare the speed of running this application with the speed of running the SnapshotEntry application.

23.7 Write a program that uses method notify to track all Entries written into a JavaSpaces service. Can you tell whether there are any Entries that the listener does not track?

23.8 In our examples, we use class JavaSpaceFinder to find a JavaSpaces service and class TransactionManagerFinder to find a transaction manager. These two classes are similar. Write a program called ServiceFinder that takes the Class object (i.e., JavaSpace.class) as input and returns the service object (i.e., JavaSpace object). Does ServiceFinder return the same JavaSpace object if you have multiple JavaSpaces services running? If not, how can you fix it so that ServiceFinder always returns the JavaSpaces service you wanted? [Hint: set the name
for each JavaSpaces service instance using the `com.sun.jini.outrigger.spaceName` property when running the JavaSpaces service from the command line. This property will register the corresponding JavaSpace stub in the Jini lookup service with the Name Entry set to the value specified.

23.9 Write a program that reads all matching Entrys from a JavaSpaces service without removing them from the JavaSpaces service.

WORKS CITED

BIBLIOGRAPHY
Li, S., Professional Jini, Birmingham, UK: Wrox Press Ltd. 2000
Objectives

- To understand the JMX technology architecture.
- To understand the design pattern of the standard MBeans.
- To make a resource manageable by defining a management interface for the resource and exposing this interface through MBeans.
- To understand the JMX agent architecture.
- To be able to design and develop management agents to expose MBeans.
- To be able to develop management applications to interact with management agents.

*Imagination is a powerful agent for creating, as it were, a second nature out of the material supplied to it by actual nature.*
Immanuel Kant

*There’s nothing remarkable about it. All one has to do is hit the right keys at the right time and the instrument plays itself.*
Johann Sebastian Bach

*Facts are stupid until brought into connection with some general law.*
Louis Agassiz
24.1 Introduction

Networks play an important role in today’s business. Businesses increasingly need networks that provide services which are customized to customer demands, consistently available and easily updated. As more organizations add networks and expand existing networks to increase productivity, network management becomes more difficult. Furthermore, every day more devices are network capable. The proper functionality of printers, network routers and other devices—many of which play an integral role in the productivity of a firm—has increased in importance. As more devices are networked and as networks grow larger, more problems can occur. Existing tools for managing networked devices often use proprietary protocols and management tools. The diversity of such protocols and tools make managing a diverse network difficult.

Another problem that plagues network management is that many existing network management schemes are inflexible and non-automated. This usually means that large amounts of time and resources are spent trying to fix and operate large networks. New technologies are required that will help shift the burden of routine network management issues to the network management software itself and leave special network maintenance issues to the network manager.

Most current network-management solutions distribute management responsibilities among several autonomous programs called agents. Typical agents pass information from the network resources to the management application and control the network resources based on commands from the management application. Only the management application makes decisions in the agent-based paradigm. These types of agents—called static agents—cannot respond on their own to problems that occur within the network. An SNMP (Simple Network Management Protocol) agent is a good example. An SNMP agent is a static agent designed to handle communication between a manageable network device and a management application. For these agent types, all managed resources must be predefined during development. This makes extending management applications to manage new resources difficult. Agents, such as the SNMP agents, lack management intelligence, because they do not have the capacity to respond to problems as they occur.
Recent technological advances have provided network management developers with the tools necessary to develop smart agents that are capable of assuming various responsibilities in a network management solution. These technological developments allow agents to incorporate themselves into frameworks in which numerous agents interact with each other and provide a dynamic and extensible network management solution.

The Java Management Extensions (JMX), developed by Sun and network-management industry leaders, define a component framework that enables developers to build automated, intelligent and dynamic network-management solutions. JMX defines a three-level management architecture—*instrumentation level*, *agent level* and *manager level* (Fig. 24.1).¹ The instrumentation level makes any Java-based object manageable. The agent level provides management services that expose the managed resources for management. The manager level allows management application access to managed resources and operates these resources via the JMX agents. In addition, JMX provides support for existing management protocols (such as SNMP) so JMX can integrate itself with existing management solutions.

With these design specifications, JMX provides:

1. platform independence, through of the Java programming language.
2. protocol independence, through support for multiple protocols.
3. reusable code, because managed resources follow *JavaBeans* component guidelines.
4. intelligent agents, because JMX allows agents to respond to resources directly, without having to wait for decisions from a resource manager.
5. scalability, because the agents are deployed in any device that runs a JVM.

---

¹ Sun Microsystems, Inc. “JMX Instrumentation and Agent Specification v1.0,” July 2000 <jcp.org/aboutJava/communityprocess/final/jsr003/index.html>
24.2 Installation

Many implementations of the JMX standard exist. One such implementation is Sun’s *Java Dynamic Management Kit (JDMK)*. JMX provides a standard API for building network management applications, but there are numerous vendors that provide additional features not specified in the JMX standard. Each vendor implementation offers advantages when solving different management problems.

To build dynamic network management solutions using JDMK, you must first download the Java Dynamic Management Kit. The Java Dynamic Management Kit 4.2 evaluation software is available at:

```
www.sun.com/software/java-dynamic/try.html
```

The Java Dynamic Management Kit is compatible with Solaris SPARC, Linux, Windows 2000 and Windows NT platforms. To download JDMK for Windows 2000 and Linux, select the link for “Other Platforms.” Download the appropriate file for your platform.

The download for Windows 2000 is a zip file. Unzipping the file creates the `SUN-Wjdmk` directory, which contains directories and files needed for developing management solution with JDMK. The files `jdmkrt.jar` and `jdmktk.jar` (located in the `SUN-Wjdmk\jdmk4.2\1.2\lib` directory) must be added to the `CLASSPATH` to compile and execute the examples.

24.3 Case Study

In the following sections, we develop a management solution using the Java Dynamic Management Kit. A complete management solution includes managed resources, the management agent and the management application. In this case study, the managed resource is a printer. Not everyone has access to a printer with management capabilities, so we developed a printer simulator. The printer is manageable by instrumenting it as an *MBean*—a manageable JavaBean component. The management application manages the printer through the *MBeanServer*—a registry for *MBeans*. The RMI connector service, provided by the JDMK, makes remote management possible. Figure 24.2 shows the architecture of the management solution.

Section 24.3.1 describes how to make a Java object manageable by exposing the managed resource. Section 24.3.2 demonstrates how to create a JMX management agent with an *MBeanServer* and the RMI connector service provided by JDMK. Section 24.3.3 describes how the management agent receives notifications sent by the managed resource. Section 24.3.4 develops a management application to interact with the management agent. Section 24.3.5 explains how to compile and run the example.

24.3.1 Instrument Resources

*Instrumentation* makes resources manageable. Resources can be devices, applications, or any Java-based objects that need to be managed through a management application. Instrumenting a resource exposes the resource’s *management interface* so that managers can communicate with the resource. The management interface contains properties that describe the resource and operations for managing the resource.
To instrument a resource, the simplest and fastest way is to write a standard MBean. MBeans, also called Managed Beans, are Java objects that instrument the resources to be managed. A standard MBean has two parts—an MBean interface and a Java class that implements the MBean interface (called an MBean class). A standard MBean must follow design patterns defined by the Java Management Extensions to standardize the instrumentation of managed resources. The design patterns for the MBean interface are as follows:

1. The MBean interface name must be composed of the MBean implementation’s Java class name followed by the suffix MBean.
2. The MBean interface specifies all get and set methods that expose the properties of the management interface.
3. Each property can have at most one set method and one get method in the management interface.
4. The MBean interface declares all operations available through the management interface. An operation is a public method whose name does not begin with get, is or set. Only public methods are exposed.

Another type of MBean is a dynamic MBean that only exposes its methods at runtime. Dynamic MBeans allow developers to change MBean behaviors at runtime.
5. Return values and parameters for the set and get methods must be serializable objects.

The design patterns for the MBean class are as follows:

1. The MBean class must implement its corresponding MBean interface.
2. The MBean class must be public and concrete (non-abstract) so that it can be instantiated.
3. The MBean class must have at least one public constructor.

Interface PrinterMBean (Fig. 24.3) is the MBean interface for our simulated printer. The MBean interface declares printer features that will be manageable. This example exposes only a small number of the printer’s manageable resources. The number of manageable resources exposed through the MBean interface relies heavily on the particular management solution being developed. The PrinterMBean interface defines three get methods whose names begin with is (lines 11–17) for Boolean properties, four get methods whose names begin with get (lines 20–29) and one set method whose name begins with set (line 32). In addition, three operations are defined—replenish the paper tray (line 35), cancel pending print jobs (line 38) and start the printing process (line 41).

```
// Fig. 24.3: PrinterMBean.java
// This interface specifies the methods that will be implemented
// by Printer, which will function as an MBean.

// deitel package
package com.deitel.advjhtpl.jmx.PrinterManagement;

public interface PrinterMBean {

    // is it printing?
    public Boolean isPrinting();

    // is it online?
    public Boolean isOnline();

    // is paper jammed?
    public Boolean isPaperJam();

    // returns paper amount in tray
    public Integer getPaperTray();

    // returns ink level in toner cartridge
    public Integer getToner();

    // returns ID of print job that is currently printing
    public String getCurrentPrintJob();

    // returns array of all queued up print jobs
    public String[] getPendingPrintJobs();
}
```

Fig. 24.3 Defining the PrinterMBean interface that exposes the printer’s management capabilities (part 1 of 2).
Interface PrinterEventListener (Fig. 24.4) defines the methods a printer-event listener must implement. The interface defines three events—out of paper, low toner and paper jam.

Class Printer (Fig. 24.5) is the implementation of interface PrinterMBean. It shows how to implement and instrument this particular MBean.

```java
31 // sets availability status of printer
32 public void setOnline( Boolean online );
33 // fills up paper tray again with paper
34 public void replenishPaperTray();
35 // cancel pending print jobs
36 public void cancelPendingPrintJobs();
37 // start printing process
38 public void startPrinting();
39 } // Printer
```

Fig. 24.4 Defining an event listener for the printer to handle out-of-paper, low-toner, and paper-jam events.

```
1 // Fig. 24.4: PrinterEventListener.java
2 // The listener interface for printer events.
3 // deitel package
4 package com.deitel.advjhtpl.jmx.Printer;
5 6 public interface PrinterEventListener { 7 public void outOfPaper();
8 public void lowToner();
9 public void paperJam();
10 }
```

Fig. 24.3 Defining the PrinterMBean interface that exposes the printer’s management capabilities (part 2 of 2).

Interface PrinterEventListener (Fig. 24.4) defines the methods a printer-event listener must implement. The interface defines three events—out of paper, low toner and paper jam.

Class Printer (Fig. 24.5) is the implementation of interface PrinterMBean. It shows how to implement and instrument this particular MBean.

```
1 // Fig. 24.5: Printer.java
2 // This class provides implementation for interface PrinterMBean
3 // and registers a managing MBean for the Printer device,
4 // which is simulated by PrinterSimulator.java.
5 // deitel package
6 package com.deitel.advjhtpl.jmx.PrinterManagement;
7 8 // Java core package
9 import java.lang.Thread;
```

Fig. 24.5 Printer MBean implementation class that represents the management contact point for all applications wishing to manage the printer (part 1 of 7).
```java
import java.util.ArrayList;

// JMX core packages
import javax.management.*;

// Deitel packages
import com.deitel.advjhtpl.jmx.Printer.*;

public class Printer implements PrinterMBean, PrinterEventListener {
    private PrinterSimulator printerSimulator;
    private static final int PAPER_STACK_SIZE = 50;
    private ObjectInstance eventBroadcasterInstance;
    private ObjectName eventBroadcasterName;
    private ObjectName printerName;
    private MBeanServer mBeanServer;

    public Printer() {
        // connect to the printer device
        printerSimulator = new PrinterSimulator( this );
        Thread myThread = new Thread( printerSimulator );
        myThread.start();

        // find all MBean servers in current JVM
        ArrayList arrayList = MBeanServerFactory.findMBeanServer( null );

        // retrieve the MBeanServer reference
        if ( arrayList.size() == 0 )
            System.out.println( "Cannot find a MBeanServer!" );
        else {

            // get the MBeanServer that has the
            // PrinterEventBroadcaster MBean registered with it
            for ( int i = 0; i < arrayList.size(); i++ ) {
                MBeanServer foundMBeanServer =
                    ( MBeanServer ) arrayList.get( i );

                // obtain the object name for the
                // PrinterEventBroadcaster MBean
                try {
                    String name = foundMBeanServer.getDefaultDomain()
                        + ":type=PrinterEventBroadcaster";
                    eventBroadcasterName = new ObjectName( name );
                }
            }
        }
    }
}
```

Fig. 24.5  **Printer** MBean implementation class that represents the management contact point for all applications wishing to manage the printer (part 2 of 7).
// handle exception when creating ObjectName
catch ( MalformedURLException exception ) {
    exception.printStackTrace();
}

// check whether the PrinterEventBroadcaster MBean is
// registered with this MBeanServer
if ( foundMBeanServer.isRegistered( eventBroadcasterName ) ) {
    mBeanServer = foundMBeanServer;
    break;
}

} // end for loop

} // end if-else to get the MBeanServer reference

} // end PrinterSimulator constructor

// will stop the printer thread from executing
// once execution should stop.
public void stop()
{
    printerSimulator.stop();
}

// Is it printing?
public Boolean isPrinting()
{
    return new Boolean( printerSimulator.isPrinting() );
}

// is online?
public Boolean isOnline()
{
    return printerSimulator.isOnline();
}

// is paper jammed?
public Boolean isPaperJam()
{
    return printerSimulator.isPaperJam();
}

// is paper tray empty?
public Integer getPaperTray()
{
    return printerSimulator.getPaperTray();
}

Fig. 24.5  Printer MBean implementation class that represents the management
contact point for all applications wishing to manage the printer (part 3 of 7).
// is toner low?
public Integer getToner()
{
    return printerSimulator.getToner();
}

// returns ID of print job that is currently printing
public String getCurrentPrintJob()
{
    return printerSimulator.getCurrentPrintJob();
}

// returns array of all queued up print jobs
public String[] getPendingPrintJobs()
{
    return printerSimulator.getPendingPrintJobs();
}

// sets status availability of printer
public void setOnline( Boolean online )
{
    if ( online.booleanValue() == true )
        printerSimulator.setOnline();
    else
        printerSimulator.setOffline();
}

// fills up the paper tray again with paper.
public void replenishPaperTray()
{
    printerSimulator.replenishPaperTray ( Printer.PAPER_STACK_SIZE );
}

// cancel pending print jobs
public void cancelPendingPrintJobs()
{
    printerSimulator.cancelPendingPrintJobs();
}

// start the printing process
public void startPrinting()
{
    printerSimulator.startPrintingProcess();
}

// send out of paper event to JMX layer
protected void fireOutOfPaperEvent()
{
    // construct parameters and signatures
    Object[] parameter = new Object[ 1 ];
parameter[ 0 ] = new Notification(
        "PrinterEvent.OUT_OF_PAPER", this, 0L );
String[] signature = new String[ 1 ];
signature[ 0 ] = "javax.management.Notification";

// invoke notification
try {
    mBeanServer.invoke( eventBroadcasterName,
        "sendNotification", parameter, signature );
} }

// handle exception when invoking method
catch ( ReflectionException exception ) {
    exception.printStackTrace();
}

// handle exception when communicating with MBean
catch ( MBeanException exception ) {
    exception.printStackTrace();
}

// handle exception if MBean not found
catch ( InstanceNotFoundException exception ) {
    exception.printStackTrace();
}
}
}

// end method outOfPaperEvent

// send low toner event to JMX layer
protected void fireLowTonerEvent()
{
    // construct parameters and signatures
    Object[] parameter = new Object[ 1 ];
    parameter[ 0 ] = new Notification(
        "PrinterEvent.LOW_TONER", this, 0L );
    String[] signature = new String[ 1 ];
signature[ 0 ] = "javax.management.Notification";

    // invoke notification
    try {
        mBeanServer.invoke( eventBroadcasterName,
            "sendNotification", parameter, signature );
    }

    // handle exception when invoking method
    catch ( ReflectionException exception ) {
        exception.printStackTrace();
    }

    // handle exception when communicating with MBean
    catch ( MBeanException exception ) {
        exception.printStackTrace();
    }

    // handle exception if MBean not found
    catch ( InstanceNotFoundException exception ) {
        exception.printStackTrace();
    }
}

Fig. 24.5 Printer MBean implementation class that represents the management contact point for all applications wishing to manage the printer (part 5 of 7).
// handle exception communicating with MBean
catch ( MBeanException exception ) {
    exception.printStackTrace();
}

// handle exception if MBean not found
catch ( InstanceNotFoundException exception ) {
    exception.printStackTrace();
}

} // end method lowTonerEvent

// send paper jam event to JMX layer
protected void firePaperJamEvent()
{
    // construct parameters and signatures
    Object[] parameter = new Object[ 1 ];
    parameter[ 0 ] = new Notification(
        "PrinterEvent.PAPER_JAM", this, 0L );
    String[] signature = new String[ 1 ];
    signature[ 0 ] = "javax.management.Notification";

    // invoke notification
    try {
        mBeanServer.invoke( eventBroadcasterName,
            "sendNotification", parameter, signature );
    }

    // handle exception when invoking method
    catch ( ReflectionException exception ) {
        exception.printStackTrace();
    }

    // handle exception communicating with MBean
    catch ( MBeanException exception ) {
        exception.printStackTrace();
    }

    // handle exception if MBean not found
    catch ( InstanceNotFoundException exception ) {
        exception.printStackTrace();
    }

} // end method paperJamEvent

// interface implementation
public void outOfPaper()
{
    // delegate call
    fireOutOfPaperEvent();
}

Fig. 24.5 Printer MBean implementation class that represents the management
contact point for all applications wishing to manage the printer (part 6 of 7).
In the `Printer` constructor, a `Printer` first connects to a printer device (lines 32–34). Typically, when managing networked resources, a connection must be established to the networked device. Because the managed resource in our example exists in the same machine as a printer simulator, we do not include the code that would establish a network connection with the printer resource. Lines 36–38 find all MBean servers in the current JVM. Method `findMBeanServer` of class `MBeanServerFactory` (package `javax.management`) provides a way to find references to MBean servers that have been instantiated in the current JVM. Method `findMBeanServer` takes one argument—a `String` that specifies the agent of the MBean server. If `null` is passed to method `findMBeanServer`, it returns all MBean servers in the JVM.

Lines 48–73 retrieve the `MBeanServer` reference that has the `PrinterEventBroadcaster` (discussed in Fig. 24.10) MBean registered with it. An `ObjectName` uniquely identifies an MBean in the MBean server. Lines 55–57 define the `ObjectName` for the `PrinterEventBroadcaster` MBean. The `ObjectName` constructor takes one argument—an arbitrary `String` that uniquely identifies the MBean within the `MBeanServer` instance. The input string must follow the format `domainName:property=value` pairs separated by commas.3

The `ObjectName` is any `String` that describes a location to which one or more MBeans may belong. This helps organize MBeans into different categories. For example, a `domainName` may be anything, such as "MyBuilding", "Boston", "College-Printers", etc... These categories can be defined such that all managed devices of a specific type on the same network are included in the same `domainName` or such that all networked devices in a particular area are grouped together. MBeans with different `domainNames` may register with the same MBean server. JMX provides an MBean pattern matching mechanism that allows users to single out MBeans that match specific `domainNames` or other search criteria. The pattern matching mechanism requires that `*` represents zero or more characters, `?` represents one or more characters and any other characters appearing in the string are matched literally.

---

3. The `String` may contain any characters except `* ? , : =`
When categorizing MBeans is not of great importance, a default domainName can be used. This domainName can be obtained from method getDefaultDomain of interface MBeanServer. Lines 67–68 check whether the PrinterEventBroadcaster MBean is registered with the retrieved MBeanServer. If so, the loop terminates.

Lines 86–102 implement the “is” getter methods defined in the PrinterMBean interface. Lines 104–126 implement the “get” getter methods defined in the MBean interface. Lines 128–135 implement the setter method defined in the MBean interface. Lines 137–154 implement the operations defined in the MBean interface. Each of the operations interacts with the printer simulator in Fig. 24.6.

Lines 156–253 handle delegated calls from the method implementations of interface PrinterEventListener (lines 255–274). Three events could happen to the printer—the out-of-paper event, the low-toner event and the paper-jam event. Lines 168–169 propagate the out-of-paper event to the management agent, lines 201–202 propagate the low-toner event to the management agent and lines 234–235 propagate the paper-jam event to the management agent. In each case, the notification of the event occurs by invoking the method sendNotification of PrinterEventBroadcaster MBean. The MBeanServer’s method invoke performs this invocation. Method invoke takes four arguments—an ObjectName that specifies the MBean on which the method is invoked (PrinterEventBroadcaster MBean), a String that specifies the method to be invoked ("sendNotification"), an array of Objects that indicates the parameters to be set when the method is invoked and an array of Strings that indicates the signature of the operation. Method invoke throws three exceptions—ReflectionException, MBeanException and InstanceNotFoundException.

Class PrinterSimulator (Fig. 24.6) mimics the printer device to which Printer (Fig. 24.5) connects.

```java
// Fig. 24.6: PrinterSimulator.java
// This class simulates a printer device on a network.

// deitel package
package com.deitel.advjhtp1.jmx.Printer;

// java core package
import java.util.Stack;

public class PrinterSimulator implements Runnable {

  private Stack printerStack = new Stack();
  private boolean isOnline = true;
  private boolean isPrinting = false;
  private boolean isPaperJam = false;

  // 50 sheets of paper in tray
  private Integer paperInTray = new Integer( 50 );

  // 100% full of ink
  private Integer tonerCartridge = new Integer( 100 );

  // Fig. 24.6 Printer simulation class capable of triggering three events (part 1 of 6).
```
private String currentPrintJob;
private boolean isAlive = true;
private PrinterEventListener eventListener;

// default public constructor
public PrinterSimulator(
    PrinterEventListener listener )
{
    eventListener = listener;
}

// stops execution of thread
public void stop()
{
    isAlive = false;
}

// main life-cycle of the printer.
// prints one job from print job stack
// 1) if offline, it pauses and waits.
// 2) if online, handles one print job
public void run()
{
    // main loop within thread
    while ( isAlive ) {
        // pause if offline
        if ( !isOnline ) {
            synchronized ( this ) {
                // waits for printer become online
                try {
                    wait();
                }
                // if interrupt occurs
                catch ( InterruptedException exception ) {
                    exception.printStackTrace();
                    System.exit( -1 );
                }
            } // end synchronized
        } // end if
        // prints one job from print job stack
        startPrintingProcess();
    } // end while
} // end while

Fig. 24.6 Printer simulation class capable of triggering three events (part 2 of 6).
```java
public void startPrintingProcess()
{
    // warm up the printer, print top print job from print
    // stack and adjust paper values and toner values
    try {
        // warm up printer for incoming batch of print jobs
        Thread.sleep( 1000 * 5 );

        if ( ( paperInTray.intValue() > 0 ) &&
        ( tonerCartridge.intValue() > 10 ) &&
        ( !isPaperJam ) ) {

            // start the printing process
            currentPrintJob = getNextPrintJob();
            isPrinting = true;

            // 12 seconds to print a normal document
            Thread.sleep( 1000 * 12 );

            // each print job uses 10 pages
            updatePaperInTray( paperInTray.intValue() - 10 );
            updateToner();
            updatePaperJam();
            isPrinting = false;

            // make sure no references are left dangling
            currentPrintJob = null;
        }
    }

    // if interrupt occurs
    catch ( InterruptedException exception ) {
        exception.printStackTrace();
        System.exit( -1 );
    }
}
```

Fig. 24.6 Printer simulation class capable of triggering three events (part 3 of 6).
// update amount of paper in paper tray
public synchronized void updatePaperInTray( int newValue )
{
    paperInTray = new Integer ( newValue );

    // fire event if paper tray low
    if ( paperInTray.intValue() <= 0 ) {
        eventListener.outOfPaper();
    }
}

// is paper jammed?
public Boolean isPaperJam()
{
    return new Boolean( isPaperJam );
}

// cancel pending print jobs
public void cancelPendingPrintJobs()
{
    synchronized ( printerStack ) {
        printerStack.clear();
    }
}

// update amount of toner available in toner cartridge
public synchronized void updateToner()
{
    // after every print job, toner levels drop 1%
    tonerCartridge = new Integer ( tonerCartridge.intValue() - 1 );

    // fire event if toner is low
    if ( tonerCartridge.intValue() <= 10 ) {
        eventListener.lowToner();
    }
}

public synchronized void updatePaperJam()
{
    if ( Math.random() > 0.9 ) {
        isPaperJam = true;
        eventListener.paperJam();
    }
}

// returns number of pages in paper tray
public synchronized Integer getPaperTray()
{
    return paperInTray;
}
public synchronized Integer getToner()
{
    return tonerCartridge;
}

public void populatePrintStack()
{
    int numOfJobs = (int) (Math.random() * 10) + 1;
    synchronized (printerStack)
    {
        for (int i = 0; i < numOfJobs; i++) {
            printerStack.add("PRINT_JOB_ID "+i);
        }
    }
}

public String getNextPrintJob()
{
    if (printerStack.isEmpty()) {
        populatePrintStack();
        try {
            Thread.sleep((int) (Math.random() * 1000 * 10));
        }
        catch (InterruptedException exception) {
            exception.printStackTrace();
            System.exit(-1);
        }
    }
    synchronized (printerStack) {
        printJob = (String) printerStack.pop();
    }
    return printJob;
} // end method getNextPrintJob

Fig. 24.6 Printer simulation class capable of triggering three events (part 5 of 6).
Fig. 24.6  Printer simulation class capable of triggering three events (part 6 of 6).

The PrinterSimulator constructor (lines 28–32) takes as an argument the PrinterEventListener to which PrinterSimulator should send events. In this example, PrinterEventListener will be the gateway through which Printer-
Simulator communicates events to the Printer MBean. As mentioned before, real scenarios would make use of other communication protocols, such as SNMP or HTTP, to facilitate communications between the managed resource on the network and the corresponding MBean. Method stop (lines 35–38) sets the printer off-line.

Method run (lines 44–72) defines the life-cycle of PrinterSimulator, which will be associated with a Thread object in the simulation. If the printer is online, run calls method startPrintingProcess (lines 74–110) to mimic the printer functionality.

Method startPrintingProcess takes five seconds to warm up the printer. If the paper tray is not empty, toner is not low and no paper is jammed, the printer starts printing the next print job. Line 91 assumes each print job takes twelve seconds. Lines 94–96 call methods to reset the properties of the printer.

Method getCurrentPrintJob (lines 113–116) returns the current print job. Method isOnline (lines 119–122) returns true if the printer is online. Method updatePaperTray (lines 125–133) updates the amount of paper in the paper tray. If no paper is available in the paper tray, it fires an out-of-paper event. Method isPaperJam (lines 136–139) returns true if the paper is jammed. Method cancelPendingPrintJobs (lines 142–147) removes pending print jobs.

Method updateToner (lines 150–160) updates the amount of toner available in the toner cartridge. If the toner is low, it fires a low-toner event. Method updatePaperJam (lines 162–168) fires a paper-jam event if the paper is jammed.

Method getPaperTray (lines 171–174) returns the number of pages in the paper tray. Method getToner (lines 177–180) returns the toner level of the printer. Method populatePrintStack (lines 183–193) generates print jobs. Method getNextPrintJob (lines 197–224) returns the next print job and generates new print jobs if the print job stack is empty. Method getPendingPrintJobs (lines 227–243) returns a list of print jobs to be printed. Method setOnline (lines 246–254) sets the printer status to online. Method setOffline (lines 257–260) sets the printer status to offline. Method replenishPaperTray (lines 264–267) fills the paper tray with the specified amount of paper. Method isPrinting (lines 270–273) returns true if the printer is printing.

24.3.2 Implementation of the JMX Management Agent

The JMX management agent acts as a bridge between the MBeans and the management application. A JMX agent typically contains an MBean server, a set of MBeans that represent the managed resources and at least one protocol adaptor or connector—MBeans that allow remote management applications to access the MBeanServer. An MBeanServer instance behaves as a registry for all MBeans. The MBeans representing the manageable resources are controlled by the management application through the MBean server. Figure 24.7 shows the JMX Agent Architecture.

MBeans represent the managed resources or management services; they are registered with the MBean server. The protocol adaptor or connector provides a gateway that allows remote applications to interact with the MBean server’s registered MBeans. Because connectors are MBeans and all MBeans can interact with each other through the MBean server, the JMX Agent Architecture allows either local or remote management applications access to MBean properties and public operations. Local management applications can manipulate MBeans directly through the MBean server; remote management applications can manipulate MBeans indirectly through connector or adapter MBeans.
Application **PrinterManagementAgent** (Fig. 24.8) is a JMX management agent. **PrinterManagementAgent** creates an **MBeanServer** and starts an RMI connector MBean service to act as a connector, starts a **PrinterEventBroadcaster** MBean that broadcasts notifications sent by the printer and starts a **Printer** MBean that is the communication bridge between the **PrinterSimulator** and the MBean server. Method `createMBeanServer` of class **MBeanServerFactory** creates a new MBean server. Lines 21–22 create an MBean server with a default domain name provided by the **MBeanServerFactory**. Lines 29–30 create an **RMICConnectorServer** by calling method `createMBean` of interface **MBeanServer**. Method `createMBean` of interface **MBeanServer** takes two arguments—a **String** that specifies the class name to be instantiated and an **ObjectName** object that specifies the name of the MBean. Method `createMBean` returns an **ObjectInstance** that represents the MBean. The string that specifies the class name cannot be **null**. If **ObjectName** is **null**, the RMI-ConnectorServer MBean—the MBean responsible for handling remote RMI connections—can provide the object name automatically. Lines 33–40 instantiate and register a **PrinterEventBroadcaster** MBean in the MBean server. Lines 43–50 instantiate and register the **Printer** MBean in the MBean server. Lines 55–77 catch the exceptions that may occur when creating the MBean and defining the object names of the MBeans. Lines 88–90 invoke method `setPort` of the **RMICConnectorServer** MBean to set the RMI connector’s port number to **5555**. Lines 91–93 invoke method `start` of the **RMICConnectorServer** MBean to start the connector. Lines 97–109 catch the exceptions that may occur when invoking the methods of the MBean.
// JMX core packages
import javax.management.*;

public class PrinterManagementAgent {
  public static void main( String[] args )
  {
    ObjectInstance rmiConnectorServer = null;
    ObjectInstance printer = null;
    ObjectInstance broadcaster = null;
    ObjectName objectName = null;

    // create an MBeanServer
    MBeanServer server =
        MBeanServerFactory.createMBeanServer();

    // create an RMI connector service, a printer simulator
    // MBean and a broadcaster MBean
    try {
      // create an RMI connector server
      rmiConnectorServer = server.createMBean (
        "com.sun.jdmk.comm.RmiConnectorServer", null);

      // create a broadcaster MBean
      String name = server.getDefaultDomain()
          + ":type=" + "PrinterEventBroadcaster";
      String className = "com.deitel.advjhtp1.jmx." +
          "PrinterManagement.PrinterEventBroadcaster";
      objectName = new ObjectName( name );
      printer = server.createMBean( className, objectName );

      // create a printer simulator MBean
      name = server.getDefaultDomain()
          + ":type=" + "Printer";
      className = "com.deitel.advjhtp1.jmx." +
          "PrinterManagement.Printer";
      broadcaster = server.createMBean( className, objectName );
    }
  }

  // handle class not JMX-compliant MBean exception
  catch ( NotCompliantMBeanException exception ) {
    exception.printStackTrace();
  }

  // handle MBean constructor exception
  catch ( MBeanException exception ) {

exception.printStackTrace();
}

// handle MBean already exists exception
catch (InstanceAlreadyExistsException exception) {
    exception.printStackTrace();
}

// handle MBean constructor exception
catch (ReflectionException exception) {
    exception.printStackTrace();
}

// handle invalid object name exception
catch (MalformedObjectNameException exception) {
    exception.printStackTrace();
}

// set port number
Object[] parameter = new Object[1];
parameter[0] = new Integer(5555);
String[] signature = new String[1];
signature[0] = "int";

// invoke method setPort on RmiConnectorServer MBean
// start the RMI connector service
try {
    server.invoke(
        rmiConnectorServer.getObjectName(), "setPort",
        parameter, signature);
    server.invoke(
        rmiConnectorServer.getObjectName(), "start",
        new Object[0], new String[0]);
}

// handle exception when executing method
catch (ReflectionException exception) {
    exception.printStackTrace();
}

// handle exception communicating with MBean
catch (MBeanException exception) {
    exception.printStackTrace();
}

// handle exception if MBean not found
catch (InstanceNotFoundException exception) {
    exception.printStackTrace();
}

} // end method main
24.3.3 Broadcasting and Receiving Notifications

This section explains how the managed resources or devices broadcast notifications via a notification broadcaster MBean and how the remote management application receives notifications sent by a notification broadcaster MBean. A notification broadcaster is an MBean that defines one or more event types. It has the ability to broadcast events to all registered listeners as the events are received from the managed resources. The notification broadcaster MBean must implement the NotificationBroadcaster interface to declare itself as a source of notifications. To receive notifications, the management application needs to contact the notification broadcasters of the manageable resource MBeans. In our case, the manageable resource is the Printer MBean, and the notification broadcaster is the PrinterEventBroadcaster MBean that is registered with the MBean server instantiated in the PrinterManagementAgent class (Fig. 24.8).

Interface PrinterEventBroadcasterMBean (Fig. 24.9) is an interface for the notification broadcaster MBean of the printer. This interface defines a single operation—sendNotification. The broadcaster is also an MBean, so we need this interface to successfully register it as a standard MBean in the MBeanServer instance.

Class PrinterEventBroadcaster (Fig. 24.10) is a standard MBean that implements the PrinterEventBroadcasterMBean interface. PrinterEventBroadcaster sends notifications of printer events to registered listeners. It extends the functionality of the javax.management.NotificationBroadcasterSupport class (line 15), which provides services such as listener registration. Lines 18–23 define three notification types. PrinterEventBroadcaster overrides method getNotificationInfo to return an array of MBeanNotificationInfo objects (package javax.management) containing the notification class name and the notification types sent. Lines 33–39 specify an array of Strings containing the notification types. Line 42 specifies the class name of the notifications. Lines 45–46 create a String that describes the notifications that PrinterEventBroadcaster MBean can broadcast. The MBeanNotificationInfo constructor (lines 49–50) takes three arguments—an array of Strings that specifies the notification types, a String that indicates the class name of the notifications and a String containing the notifications’ description. Line 52 returns the MBeanNotificationInfo array.

```java
// Fig. 24.9: PrinterEventBroadcasterMBean.java
// This class defines the MBean interface.
package com.deitel.advjhtp1.jmx.PrinterManagement;

import javax.management.Notification;

public interface PrinterEventBroadcasterMBean {
    public void sendNotification( Notification notification );
}
```

Fig. 24.9 Notification broadcaster MBean interface.
Class **PrinterEventHandler** (Fig. 24.11) receives event notifications from the broadcaster MBean. Lines 24–53 define an anonymous inner class implementing the notification listener interface (**javax.management.NotificationListener**). The implementation of method **handleNotification** (lines 26–51) handles incoming events. Lines 33–37 handle the out-of-paper event. Lines 39–43 handle the low-toner event. Lines 45–49 handle the paper-jam event. Line 64 of the **PrinterEventHandler** constructor (lines 56–86) sets the notification forwarding mode to **ClientNotificationHandler.PUSH_MODE** (package **com.sun.jdmk.comm**). In push mode, as soon as the RMI connector server receives a notification, it forwards the notification to the RMI connector client. Lines 68–73 add a notification listener to the broadcaster MBean. Lines 68–70 specify the object name of the broadcaster MBean. Method **addNotificationListener** (lines 72–73) takes four arguments—an **ObjectName** that specifies the name of the broadcaster MBean on which the notification listener will be registered, a **NotificationListener** that handles the notifications sent by the broadcaster MBean, a class that implements interface **NotificationFilter** to filter the notification and an **Object** that contains the context to be sent to the listener. In our example, the **NotificationListener** (specified in lines 24–53) is registered with the **PrinterEventBroadcaster** MBean that is connected to the MBean server. Lines 76–84 catch the exceptions that may occur when adding the listener. Method **handleOutOfPaperEvent** (lines 89–92) delegates the out-of-paper event. Method **handleLowTonerEvent** (lines 95–98) delegates the low-toner event. Method **handlePaperJamEvent** (lines 101–104) delegates the paper-jam event.

```
1 // Fig. 24.10: PrinterEventBroadcaster.java
2 // This class defines an MBean that
3 // provides events information.
4
5 // deitel package
6 package com.deitel.advjhtp1.jmx.PrinterManagement;
7
8 // JMX core packages
9 import javax.management.MBeanNotificationInfo;
10 import javax.management.NotificationBroadcasterSupport;
11
12 // extends NotificationBroadcasterSupport to adopt its
13 // functionality.
14 public class PrinterEventBroadcaster
15 extends NotificationBroadcasterSupport
16 implements PrinterEventBroadcasterMBean {
17
18    private static final String OUT_OF_PAPER =
19        "PrinterEvent.OUT_OF_PAPER";
20    private static final String LOW_TONER =
21        "PrinterEvent.LOW_TONER";
22    private static final String PAPER_JAM =
23        "PrinterEvent.PAPER_JAM";
24
```

**Fig. 24.10** Notification broadcaster MBean implementation that broadcasts events generated by the printer (part 1 of 2).
// provide information about deliverable events
public MBeanNotificationInfo[] getNotificationInfo()
{
    // array containing descriptor objects
    MBeanNotificationInfo[] descriptorArray =
        new MBeanNotificationInfo[1];

    // different event types
    String[] notificationTypes = new String[3];
    notificationTypes[0] =
        PrinterEventBroadcaster.OUT_OF_PAPER;
    notificationTypes[1] =
        PrinterEventBroadcaster.LOW_TONER;
    notificationTypes[2] =
        PrinterEventBroadcaster.PAPER_JAM;

    // notification class type
    String classType = "javax.management.Notification";

    // description of MBeanNotificationInfo
    String description =
        "Notification types for PrinterEventBroadcaster";

    // populate descriptor array
    descriptorArray[0] = new MBeanNotificationInfo(
        notificationTypes, classType, description);

    return descriptorArray;
}

Fig. 24.10 Notification broadcaster MBean implementation that broadcasts events generated by the printer (part 2 of 2).

// Fig. 24.11: PrinterEventHandler.java
// The class adds a listener to the broadcaster MBean and
// defines the event handlers when event occurs.

// deitel package
package com.deitel.advjhtpl.jmx.Client;

// JMX core packages
import javax.management.*;

// JDMK core packages
import com.sun.jdmk.comm.RmiConnectorClient;
import com.sun.jdmk.comm.ClientNotificationHandler;

// Deitel packages
import com.deitel.advjhtpl.jmx.Printer.*;

Fig. 24.11 Receiving event notifications from the MBean server and handling the printer-specific events (part 1 of 3).
public class PrinterEventHandler {
    private RmiConnectorClient rmiClient;
    private PrinterEventListener eventTarget;

    // notification listener anonymous inner class
    private NotificationListener notificationListener =
        new NotificationListener() {
            public void handleNotification(
                Notification notification, Object handback ) {
                // retrieve notification type
                String notificationType = notification.getType();

                // handle different notifications
                if ( notificationType.equals("PrinterEvent.OUT_OF_PAPER") ) {
                    handleOutOfPaperEvent();
                    return;
                }

                if ( notificationType.equals("PrinterEvent.LOW_TONER") ) {
                    handleLowTonerEvent();
                    return;
                }

                if ( notificationType.equals("PrinterEvent.PAPER_JAM") ) {
                    handlePaperJamEvent();
                    return;
                }

            } // end method handleNotification
        } // end anonymous inner class

    // default constructor
    public PrinterEventHandler(
        RmiConnectorClient inputRmiClient,
        PrinterEventListener inputEventTarget ) {
        rmiClient = inputRmiClient;
        eventTarget = inputEventTarget;

        // set notification push mode
        rmiClient.setMode( ClientNotificationHandler.PUSH_MODE );
    }

} // class PrinterEventHandler

Fig. 24.11 Receiving event notifications from the MBean server and handling the printer-specific events (part 2 of 3).
The management application in our example provides a simple graphical user interface for managing the printer. Class `ClientPrinterManagement` (Fig. 24.12) establishes a connection to the MBean server via the `RmiConnectorClient` (package `com.sun.jdmk.comm`). The management of the printer relies on the MBean server. The `ClientPrinterManagement` constructor (lines 24–39) creates a `RmiConnectorClient` corresponding to the `RmiConnectorServer` that is attached to the `MBeanServer` created by `PrinterManagementAgent` (Fig. 24.8). Line 27 creates the

```java
66   // register listener
67   try {
68       ObjectName objectName = new ObjectName(
69           rmiClient.getDefaultDomain()
70           + ":type=" + "PrinterEventBroadcaster" );
71
72       rmiClient.addNotificationListener( objectName,
73           notificationListener, null, null );
74   }
75
76   // if MBean does not exist in the MBean server
77   catch ( InstanceNotFoundException exception ) {
78       exception.printStackTrace();
79   }
80
81   // if the format of the object name is wrong
82   catch ( MalformedObjectNameException exception ) {
83       exception.printStackTrace();
84   }
85
86 } // end PrinterEventHandler constructor
87
88   // delegate out of paper event
89   private void handleOutOfPaperEvent() {
90       eventTarget.outOfPaper();
91   }
92
93   // delegate low toner event
94   private void handleLowTonerEvent() {
95       eventTarget.lowToner();
96   }
97
98   // delegate paper jam event
99   private void handlePaperJamEvent() {
100      eventTarget.paperJam();
101   }
102
103 } // end PrinterEventHandler constructor

Fig. 24.11 Receiving event notifications from the MBean server and handling the printer-specific events (part 3 of 3).

24.3.4 Management Application

The management application in our example provides a simple graphical user interface for managing the printer. Class `ClientPrinterManagement` (Fig. 24.12) establishes a connection to the MBean server via the `RmiConnectorClient` (package `com.sun.jdmk.comm`). The management of the printer relies on the MBean server. The `ClientPrinterManagement` constructor (lines 24–39) creates a `RmiConnectorClient` corresponding to the `RmiConnectorServer` that is attached to the `MBeanServer` created by `PrinterManagementAgent` (Fig. 24.8). Line 27 creates the
RmiConnectorClient. Lines 30–31 obtain the address for the connector client. Line 34 specifies the port number for the connector client. The port number of the connector client should agree with that of the connector server. Line 37 establishes a connection with the remote MBeanServer. Method getClient (lines 42–45) returns RmiConnectorClient reference. In the main method, lines 57–58 start the GUI for the management application. Lines 61–63 display the output window (shown in Fig. 24.14–Fig. 24.18).

```java
// Fig. 24.12: ClientPrinterManagement.java
// This application establishes a connection to the MBeanServer
// and creates an MBean for PrinterSimulator.

// deitel package
package com.deitel.advjhtp1.jmx.Client;

// Java core packages
import java.awt.*;
import java.awt.event.*;

// JMX core packages
import javax.management.*;

// JDMX core packages
import com.sun.jdmk.comm.RmiConnectorClient;
import com.sun.jdmk.comm.RmiConnectorAddress;

public class ClientPrinterManagement {
    private RmiConnectorClient rmiClient;

    // instantiate client connection
    public ClientPrinterManagement() {
        // create connector client instance
        rmiClient = new RmiConnectorClient();

        // create address instance
        RmiConnectorAddress rmiAddress =
                new RmiConnectorAddress();

        // specify port
        rmiAddress.setPort( 5555 );

        // establish connection
        rmiClient.connect( rmiAddress );
    } // end ClientPrinterManagement constructor

Fig. 24.12  Connecting to the MBeanServer remotely and creating a PrinterSimulator MBean (part 1 of 2).
```
Class **PrinterManagementGUI** (Fig. 24.13) defines the GUI for the management application. The GUI contains a panel for displaying the printer status and buttons for updating the status, adding paper and cancelling pending print jobs.
import com.sun.jdmk.comm.RmiConnectorClient;
import com.sun.jdmk.comm.RmiConnectorAddress;

import com.deitel.advjhtp1.jmx.Printer.*;

public class PrinterManagementGUI extends JFrame
    implements PrinterEventListener {

    private class TextAppender implements Runnable {
        private String text;
        private JTextArea textArea;

        public TextAppender( JTextArea area, String newText ) {
            text = newText;
            textArea = area;
        }

        public void run() {
            textArea.append( text );
            textArea.setCaretPosition( textArea.getText().length() );
        }
    }

    private ObjectName objectName;
    private RmiConnectorClient client;
    private JTextArea printerStatusTextArea = new JTextArea();
    private JTextArea printerEventTextArea = new JTextArea();

    public PrinterManagementGUI( RmiConnectorClient rmiClient ) {
        super( "JMX Printer Management Example" );
        Container container = getContentPane();
        JPanel printerStatusPanel = new JPanel();

        // status panel
        printerStatusPanel = new JPanel();
printerStatusPanel.setPreferredSize(new Dimension(512, 200));

JScrollPane statusScrollPane = new JScrollPane();
statusScrollPane.setAutoscrolls(true);
statusScrollPane.setPreferredSize(new Dimension(400, 150));
statusScrollPane.getViewport().add(printerStatusTextArea, null);
printerStatusPanel.add(statusScrollPane, null);

// buttons panel
JPanel buttonPanel = new JPanel();
buttonPanel.setPreferredSize(new Dimension(512, 200));

// define action for Check Status button
JButton checkStatusButton = new JButton("Check Status");
checkStatusButton.addActionListener(new ActionListener(){

    public void actionPerformed(ActionEvent event) {
        checkStatusButtonAction(event);
    }
});

// define action for Add Paper button
JButton addPaperButton = new JButton("Add Paper");
addPaperButton.addActionListener(new ActionListener(){

    public void actionPerformed(ActionEvent event) {
        addPaperButtonAction(event);
    }
});

// define action for Cancel Pending Print Jobs button
JButton cancelPendingPrintJobsButton = new JButton("Cancel Pending Print Jobs");
cancelPendingPrintJobsButton.addActionListener(new ActionListener(){

    public void actionPerformed(ActionEvent event) {
        cancelPendingPrintJobsButtonAction(event);
    }
});

// add three buttons to the panel
buttonPanel.add(checkStatusButton, null);

Fig. 24.13 GUI for the management application (part 3 of 9).
buttonPanel.add( addPaperButton, null );
buttonPanel.add( cancelPendingPrintJobsButton, null );

// events panel
JPanel printerEventPanel = new JPanel();
printerEventPanel.setPreferredSize(
        new Dimension( 512, 200 ) );
JScrollPane eventsScrollPane = new JScrollPane();
eventsScrollPane.setAutoscrolls( true );
eventsScrollPane.setPreferredSize(
        new Dimension( 400, 150 ) );
eventsScrollPane.getViewport().add(
        printerEventTextArea, null );
printerEventPanel.add( eventsScrollPane, null );

// initialize the text
printerStatusTextArea.setText( "Printer Status: ---\n" );
printerEventTextArea.setText( "Events: --- \n" );

// put all the panels together
container.add( printerStatusPanel, BorderLayout.NORTH );
container.add( printerEventPanel, BorderLayout.SOUTH );
container.add( buttonPanel, BorderLayout.CENTER );

// set RmiConnectorClient reference
client = rmiClient;

try {
    String name = client.getDefaultDomain()
        + ":type=" + "Printer";
   ObjectName = new ObjectName( name );
    client.invoke( objectName, "startPrinting",
        new Object[ 0 ], new String[ 0 ] );
}

// invalid object name
catch ( MalformedObjectNameException exception ) {
    exception.printStackTrace();
}

// if cannot invoke the method
catch ( ReflectionException exception) {
    exception.printStackTrace();
}

// if invoked method throws exception
catch ( MBeanException exception ) {
    exception.printStackTrace();
}
// if MBean is not registered with MBean server
catch ( InstanceNotFoundException exception ) {
    exception.printStackTrace();
}

// instantiate PrinterEventNotifier
PrinterEventHandler printerEventHandler =
    new PrinterEventHandler( client, this );

// unregister MBean when close the window
addWindowListener(
    new WindowAdapter() {
        public void windowClosing( WindowEvent event )
        {
            // unregister MBean
            try {

                // unregister the PrinterSimulator MBean
                client.unregisterMBean( objectName );

                // unregister the PrinterEventBroadcaster
                // MBean
                String name = client.getDefaultDomain()
                    + ":type=PrinterEventBroadcaster";
                objectName = new ObjectName( name );
                client.unregisterMBean( objectName );

            } catch ( MalformedObjectNameException exception) {
                exception.printStackTrace();
            }

            // if exception is caught from method preDeregister
            catch ( MBeanRegistrationException exception ) {
                exception.printStackTrace();
            }

            // if MBean is not registered with MBean server
            catch ( InstanceNotFoundException exception ) {
                exception.printStackTrace();
            }

            // terminate the program
            System.exit( 0 );
        }
    });
} // end method windowClosing

} // end WindowAdapter constructor

}; // end addWindowListener

} // end PrinterManagementGUI constructor

Fig. 24.13 GUI for the management application (part 5 of 9).
// out of paper events
public void outOfPaper()
{
    SwingUtilities.invokeLater(
        new TextAppender( printerEventTextArea,
             "\nEVENT: Out of Paper!\n" ) );
}

// toner low events
public void lowToner()
{
    SwingUtilities.invokeLater(
        new TextAppender( printerEventTextArea,
             "\nEVENT: Toner Low!\n" ) );
}

// paper jam events
public void paperJam()
{
    SwingUtilities.invokeLater(
        new TextAppender( printerEventTextArea,
             "\nEVENT: Paper Jam!\n" ) );
}

// add paper to the paper tray
public void addPaperButtonAction( ActionEvent event )
{
    try {
        client.invoke( objectName, "replenishPaperTray",
             new Object[ 0 ], new String[ 0 ] );
    }

    // if cannot invoke the method
    catch ( ReflectionException exception)
    { exception.printStackTrace();
    }

    // if invoked method throws exception
    catch ( MBeanException exception ) {
        exception.printStackTrace();
    }

    // if MBean is not registered with MBean server
    catch ( InstanceNotFoundException exception ) {
        exception.printStackTrace();
    }

} // end method addPaperButtonAction

Fig. 24.13 GUI for the management application (part 6 of 9).
// cancel pending print jobs
public void cancelPendingPrintJobsButtonAction(
    ActionEvent event )
{
    try {
        client.invoke( objectName, "cancelPendingPrintJobs",
            new Object[ 0 ], new String[ 0 ] );
    }
    // if cannot invoke the method
    catch ( ReflectionException exception)
    {
        exception.printStackTrace();
    }
    // if invoked method throws exception
    catch ( MBeanException exception ) {
        exception.printStackTrace();
    }
    // if MBean is not registered with MBean server
    catch ( InstanceNotFoundException exception ) {
        exception.printStackTrace();
    }
} // end method cancelPendingPrintJobsButtonAction

public void checkStatusButtonAction( ActionEvent event )
{
    Object onlineResponse = null;
    Object paperJamResponse = null;
    Object printingResponse = null;
    Object paperTrayResponse = null;
    Object pendingPrintJobsResponse = null;

    // manage printer remotely
    try {

        // check if the printer is on line
        onlineResponse = client.invoke( objectName,
            "isOnline", new Object[ 0 ], new String[ 0 ] );

        // check if the printer is paper jammed
        paperJamResponse = client.invoke( objectName,
            "isPaperJam", new Object[ 0 ], new String[ 0 ] );

        // check if the printing is pringint
        printingResponse = client.invoke( objectName,
            "isPrinting", new Object[ 0 ], new String[ 0 ] );

        // get the paper tray
        paperTrayResponse = client.invoke( objectName,
            "getPaperTray", new Object[ 0 ], new String[ 0 ] );

    } // end try
} // end method checkStatusButtonAction
// get pending print jobs
    pendingPrintJobsResponse = client.invoke( objectName,
        "getPendingPrintJobs" , new Object[ 0 ],
        new String[ 0 ] );

// if cannot invoke the method
    catch ( ReflectionException exception ) {
        exception.printStackTrace();
    }

// if invoked method throws exception
    catch ( MBeanException exception ) {
        exception.printStackTrace();
    }

// if MBean is not registered with MBean server
    catch ( InstanceNotFoundException exception ) {
        exception.printStackTrace();
    }

    // status for the online condition
    boolean isOnline =
        ( ( Boolean ) onlineResponse ).booleanValue();

    // display status
    if ( isOnline ) {
        SwingUtilities.invokeLater( new TextAppender( 
            printerStatusTextArea,
            "\nPrinter is ONLINE.\n" ) );
    }
    else {
        SwingUtilities.invokeLater( new TextAppender( 
            printerStatusTextArea,
            "\nPrinter is OFFLINE.\n" ) );
    }

    // status for the paper jam condition
    boolean isPaperJam =
        ( ( Boolean ) paperJamResponse ).booleanValue();

    // display status
    if ( isPaperJam ) {
        SwingUtilities.invokeLater( new TextAppender( 
            printerStatusTextArea,
            "Paper jammed.\n" ) );
    }
    else {
        SwingUtilities.invokeLater( new TextAppender( 
            printerStatusTextArea,
            "No Paper Jam.\n" ) );
    }

Fig. 24.13 GUI for the management application (part 8 of 9).
Lines 32–56 define a `private` inner class, `TextAppender` to append text to the Swing container in a thread-safe manner. Lines 154–158 call method `startPrinting` of the `Printer` MBean to ask the printer to start the printing process. The management application needs to handle events received from the broadcaster MBean. Lines 182–183 call the `PrinterEventHandler` (Fig. 24.11) constructor to add a notification listener to the broadcaster MBean. The constructor takes two arguments. The first argument,
**ConnectorClient**, provides a reference to the connector client through which the registration process can occur. The second argument, **PrinterEventListener**, is a reference to class **PrinterManagementGUI** that will handle all transmitted events.

Lines 186–226 install a window listener that unregisters the **Printer** MBean and the **PrinterEventBroadcaster** MBean when the user closes the output window. Line 194 unregisters the **PrinterSimulator** MBean. Lines 198–201 unregister the **PrinterEventBroadcaster** MBean. Method **unregisterMBean** (lines 194 and 201) takes one argument—an **ObjectName** that specifies the name of the MBean to be removed from the remote **MBeanServer**. Method **unregisterMBean** deletes an MBean reference from the remote **MBeanServer**.

Method **outOfPaper** (lines 231–236) displays the out-of-paper event in the event panel. Method **lowToner** (lines 239–244) displays the low-toner event in the event panel. Method **paperJam** (lines 247–252) displays the paper-jam event in the event panel. Method **addPaperButtonAction** (lines 255–278) executes when the user clicks the **Add Paper** button. Lines 258–259 invoke the **Printer** MBean’s **replenishPaperTray** operation, which fills the paper tray with 50 pages. Method **cancelPendingPrintJobsButtonAction** (lines 281–305) executes when the user clicks the **Cancel Pending Print Jobs** button. Lines 285–288 invoke the **Printer** MBean’s **cancelPendingPrintJobs** operation. Method **checkStatusButtonAction** (lines 307–425) performs when the user clicks on the **Check Status** button. Lines 318–337 invoke the **Printer** MBean’s **isOnline**, **isPaperJam**, **isPrinting**, **getPaperTray**, and **getPendingPrintJobs** get methods. Lines 355–422 prepare the outputs.

### 24.3.5 Compiling and Running the Example

Before compiling the Java code, the JAR files **jdmkrt.jar** and **jdmktk.jar** should be included in the **CLASSPATH**. Compile the files in package **com.deitel.advjhtp1.jmx.Printer** first, then compile the files in package **com.deitel.advjhtp1.jmx.PrinterManagement**. Finally compile the files in package **com.deitel.advjhtp1.jmx.Client**.

To run the example, first start the **PrinterManagementAgent**. Class **PrinterManagementAgent** creates an **MBeanServer** and initializes the RMI connector service. Start the **ClientPrinterManagement** application. Class **ClientPrinterManagement** starts the printer simulator and invokes the GUI to manage the printer. Figure 24.14 shows the initial output window.

Figure 24.15 shows the printer status by after an out-of-paper event has occurred and the user pressed the **Check Status** button.

Figure 24.16 shows the printer status by after clicking the **Add Paper** button and the **Check Status** button.

Figure 24.17 shows the printer status after a paper-jam event has occurred and the user clicked the **Check Status** button.

Figure 24.18 shows the printer status after clicking the **Cancel Pending Print Jobs** button and **Check Status** button.
**Fig. 24.14** Initial output window.

**Fig. 24.15** Printer status after an out-of-paper event occurred.
Fig. 24.16  Printer status after an add-paper action is taken.

Fig. 24.17  Printer status after a paper-jam event occurred.
24.4 Internet and World Wide Web Resources

www.sun.com/software/java-dynamic/service_driven.html
This page introduces the new generation network—service driven network.

www.sun.com/software/java-dynamic/wp_jkmk40.html
This page is the white paper for the Java Dynamic Management Kit.

www.sun.com/software/java-dynamic/wp_jini_jdmk.html
This page demonstrates how the Jini and the Java Dynamic Management Kit work together to provide spontaneous management.

www.sun.com/software/java-dynamic/qa.html
This site contains the FAQs about the Java Dynamic Management Kit.

This site contains the article Java enters the management arena with JMX and Java DMK written by Max Goff.

java.sun.com/products/JavaManagement/wp
This page contains the Java Management Extensions White Paper.

**SUMMARY**

- Today’s network management is performed through management applications via agents.
- Many existing agents are limited because they lack event handling capabilities that adapt to changing network conditions.
• The management capabilities of a static agent must be predefined during development.
• The Java Dynamic Management Kit (JDMK) provides developers of Java-based applications with means to build automated and intelligent dynamic network management solutions.
• JMX defines a three-level management architecture—instrumentation level, agent level and manager level. The instrumentation level makes any Java-based object manageable by exposing its management interface. The agent level provides management services to expose the managed resources for management. The management level allows a management application to access and operate the managed resources via the JMX agents.
• JMX implementations provide interfaces to existing management protocols so that developers can integrate new management applications with existing management solutions.
• JMX uses JavaBeans to build reusable management solutions.
• Java Dynamic Management Kit (JDMK) is one of many implementations of the JMX specification.
• To run an management application written with JDMK, the JAR files *jdmkrt.jar* and *jdmktk.jar* must be added to the CLASSPATH.
• A complete management solution includes the managed resources, the management agent and the management application.
• The task of instrumentation is to make resources manageable. Resources can be devices, applications and any Java-based objects that need to be managed through a management application.
• An MBean is used to instrument a resource (i.e., make it manageable).
• A standard MBean has two parts—the MBean interface and the Java class that implements the MBean interface (i.e., the MBean class).
• A standard MBean must follow design patterns defined by the Java Management Extensions specification to standardize the instrumentation of managed resources.
• The MBean interface must have MBean implementation’s Java class name followed by the MBean suffix.
• An operation is a *public* method whose name does not begin with *get*, *is* or *set*. Only *public* methods are exposed.
• Method *findMBeanServer* of class *MBeanServerFactory* provides a way to obtain references to MBean servers that have been instantiated in the JVM.
• Method *invoke* of interface *MBeanServer* invokes a specified method on the indicated MBean.
• Method *createMBean* of interface *MBeanServer* instantiates an MBean object and gives it a unique identifying name.
• The JMX management agent acts as a communication bridge between the MBeans and the management application.
• A JMX agent typically contains an MBean server, a set of MBeans that represent either the managed sources or the management resources, and at least one protocol adaptor or connector to allow remote management application access to the agent.
• The MBeans represent the managed resources or management services; they are registered with the MBean server.
• The local management application manipulates the MBean directly through the MBean server. The remote management application manipulates the MBean indirectly through the MBean server via either a protocol adaptor or a connector.
• Method *createMBeanServer* of class *MBeanServerFactory* creates an MBean server.
• A notification broadcaster MBean is an MBean that contains the source of notifications.
• The notification broadcaster MBean has the ability to broadcast the notifications received from the managed resources or devices to the MBean server.
• Event broadcaster classes can extend class NotificationBroadcasterSupport (package javax.management) to inherit services such as listener registration.
• In push mode, as soon as the RMI connector server receives a notification, it forwards the notification to the RMI connector client.
• Method unregisterMBean removes an MBean reference from the remote MBean server.

TERMINOLOGY
addNotificationListener of RmiConnectorClient agent JDMK
agent level JMX
ClientNotificationHandler interface MBean
connector management level
MBeanException

findMBeanServer method of MBeanServerFactory MBeanNotificationInfo
getDomain method of MBeanServerFactory MBeanServer interface
handleNotification method of NotificationListener protocol adaptor
instrumentation level ReflectionException
intelligent agent RmiConnectorClient class
Java Dynamic Management Kit (JDMK) scalability
Java Management Extensions (JMX) SNMP
JavaBeans standard MBean

SELF-REVIEW EXERCISES
24.1 Fill in the blanks in each of the following:
   a) JMX defines a three-level management architecture: _________, _________ and _________.
   b) A(n) _________ instruments a resource for management.
   c) Method _________ of class MBeanServerFactory obtains references to MBean servers instantiated in the JVM.
   d) To instantiate and register an MBean in the MBean server, call method _________ of interface MBeanServer.
   e) A JMX agent contains at least one _________ or _________ to allow remote management applications to access the agent.
   f) The notification broadcaster MBean can implement interface _________ to declare itself as a source of notifications.

24.2 State whether each of the following is true or false. If false, explain why.
   a) A property in a standard MBean can have multiple set methods.
   b) When implementing an MBean, if no public constructor is defined, the Java compiler will display an error message.
   c) Each MBean object registered in the MBean server must have a unique object name.
   d) By extending class NotificationBroadcasterSupport, a notification broadcaster MBean inherits the services of this class and does not need to implement interface NotificationBroadcaster.
ANSWERS TO SELF-REVIEW EXERCISES

24.1 a) instrumentation level, agent level, manager level. b) standard MBean. c) findMBeanServer. d) createMBean. e) protocol adaptor, connector. f) NotificationBroadcaster.

24.2 a) False. Only one setter method and one getter method are allowed for a single property in the management interface. b) False. The Java compiler provides a public, no-argument constructor to the MBean class by default. c) True. d) True. e) True.

EXERCISES

24.3 What is the purpose of connector and adaptor MBeans in JMX? How are connector and adaptor MBeans similar to other MBeans? How are they different? Compare and contrast connector MBeans to our Printer MBean.

24.4 Add an HTML protocol adaptor to the MBeanServer created in PrinterManagement-Agent.java. Run the ClientPrinterManagement application as before. Use URL http://localhost:8082 to view the MBeans that are registered with the MBean server. How many MBeans can you see? Explain why they are there. Can you distinguish the MBeans that are directly registered with the MBean server from those that are not? Use the JMX API documentation for assistance.

24.5 Try to change properties and perform operations in the Printer MBean (Fig. 24.5) from the URL given in the previous exercise. Do results correspond with those of the GUI that is invoked by the ClientPrinterManagement application?

24.6 Expand the capabilities of class PrinterSimulator (Fig. 24.6) to include a method called removePaperJam that will fix any paper jam event that occurs. Modify the GUI in Fig. 24.13 to include a button labeled Remove Paper Jam. Modify all necessary files in the printer simulator example such that when a paper jam event occurs, the user may fix the paper jam by pressing the new button.

24.7 Modify all necessary files in the printer simulator example to allow a manager to shut down and start up a printer remotely through the PrinterManagementGUI.

24.8 Modify the PrinterSimulator MBean and GUI such that the client application can perform a refillToner operation when the toner is lower than 10%.

BIBLIOGRAPHY


Objectives

- To understand the Jiro technology architecture.
- To be able to locate static services.
- To understand how the controller service, event service, log service, scheduling service and transaction service work.
- To be able to deploy dynamic services.
- To be able to instantiate dynamic services.

All love that has not friendship for its base,
Is like a mansion built upon the sand
Ella Wheeler Wilcox

You know my method. It is founded upon the observance of trifles.
Arthur Conan Doyle

Oh let us love our occupations,
Bless the squire and his relations,
Live upon our daily rations,
And always know our proper stations.
Charles Dickens
25.1 Introduction

The management of distributed resources, networks and storage devices is becoming increasingly sophisticated in response to the complexity of distributed resources, heterogeneous networks and heterogeneous storage devices. A network may include computers and devices from many different manufacturers. Computers connected to a network may have different architectures, platforms and even data formats. All of these increase the complexity and costs of managing networks.

To reduce complexity and cost, a management solution should be:

1. automated, so that the management can be monitored and controlled to minimize human response when problems occur;
2. centralized, so that distributed resources can be managed from a single location;
3. standardized, so that management software can communicate with other related management software;
4. open and interoperable, so that management software can interact with various different types of manageable resources across a network;
5. platform independent, so that cross-platform information can be managed;
6. fast to develop and easy to deploy,
7. highly available.

Sun’s Java-based Jiro™ Technology provides infrastructure to develop management solutions that can manage distributed resources over heterogeneous networks. Jiro is an implementation of the Federated Management Architecture (FMA) specification (developed under the Java Community Process) that defines an architecture for communication between heterogeneous managed resources (such as devices, systems and applications). Jiro technology uses a three-tier architecture for management solutions (Fig. 25.1). The top tier is the client tier. The client, in the top tier, locates and communicates with the management services. The middle tier is the Jiro technology that provides both static and dynamic management services. The bottom tier consists of the heterogeneous managed resources.

Besides Jiro, several other industry standards exist. Among these standards, the best known are SNMP and CIM. The Simple Network Management Protocol (SNMP) provides a standardized network-management protocol for managing devices connected to a TCP/IP network. The Common Information Model (CIM) defines a standard model for describing management information in networks. Jiro supports both standards.

Jiro does not manage resources directly. Instead, it provides both a management infrastructure and management services, which are necessary to develop a management application. The management infrastructure uses the RMI activation system, Jini and a class server. The RMI activation system activates or restarts services as they are needed. Jini provides a dynamic lookup service that enable management domains to find services and each other without prior knowledge of the lookup service’s location. The class server provides the classes downloaded dynamically by clients.

### 25.2 Installation

To develop a management application with Jiro, first ensure that you have the Java 2 SDK version 1.3.x installed. Then, download and install the Jiro Runtime Environment and the Jiro Technology Software Development Kit (Jiro SDK). The Jiro Runtime Environment and the Jiro SDK (for Windows NT/2000 and Solaris) can be downloaded from:

www.jiro.com/downloads
We discuss the Windows NT/2000 version here. The steps for the Solaris version are similar. Although downloading Jiro is free, you are required to register and accept the licence agreement first. At the time of this publication, the current version of the Jiro Runtime is v1.5 and the current version of the Jiro SDK is v1.5.

When installing the Jiro Runtime Environment, you will be asked to provide the location of Java v1.3.x. You also will be asked to provide the Jiro management domain’s name. A Jiro management domain contains the managed resources and the management services to manage those resources. The name of the Jiro management domain should be a String that uniquely identifies on the local network each shared management server (responsible for providing many of the essential services in a Jiro network). The domain name can be of any string format, as long as the string does not contain space characters. A typical representation of the domain name has the format jiro:domainName. To ensure the uniqueness of the domainName, replace our domain name with the hostname of the machine on which you install Jiro.

When installing the Jiro SDK, you will be asked to enter the location of Java v1.3.x and the Jiro management domain name. Use the same domainName as in the Jiro Runtime Environment installation.

After installing the Jiro Runtime Environment and the Jiro SDK, update the execution PATH environment variable so that it includes the path to the Jiro executables. The Jiro executables normally are located in JiroSDK\bin.

Software Engineering Observation 25.1

The Jiro management domain name should be unique within the IP multicast radius. Use the localhost name as the management domain name, to ensure uniqueness.

25.3 Starting Jiro

Before starting Jiro, the Jiro management domain name must be specified. If you did not specify the domain name at installation time, you should specify the domain name in the JiroSDK\etc\server.config file. The Jiro distribution provides a graphical user interface for configuring, starting and stopping the Jiro framework. You can execute this GUI using the file igniter.bat in the bin directory of the Jiro installation. Figure 25.2 shows the initial screen of the GUI.

Fig. 25.2 Jiro GUI: Igniter initial screen.

1. The IP multicast radius defines the "range" of the Jini multicast discovery protocols.
The five red lights indicate that Jiro is not yet running. You can start Jiro with a cleaning process by checking the Clean before start checkbox, or you can start without a cleaning process by unchecking the Clean before start checkbox. The cleaning process removes previously deployed dynamic services and resets state information from previous Jiro executions. Clicking the Start button starts Jiro with the default configuration, unless the Jiro configuration files have been modified previously. Once Jiro starts, the class server, the rmid, the transaction service, the shared Jiro station, the controller service, the log service, the event service and the scheduling service also start. We discuss these services in Section 25.6. The transaction service, the shared Jiro station, the controller service, the log service, the event service and the scheduling service register with the Jini lookup service, which also starts when the user clicks the Start button. If startup completes successfully, the five lights turn green. To view the output of the start process, go to the Options menu and check the Display Console checkbox. Figure 25.3 shows the Igniter GUI with Display Console checked after the startup process completes. Select the Errors panel to view the error messages during the startup process.

To configure the settings of Jiro, stop Jiro by clicking the Stop button if Jiro is running, then go to the File menu and select Edit Configuration. For information about configuration parameters, please refer to the Jiro Technology Installation and Configuration Guide that comes with the Jiro SDK installation. This document is located in the Jiro installation’s docs directory and is available in PDF (install_config.pdf) and postscript (install_config.ps) formats.

Software Engineering Observation 25.2

You cannot edit the Jiro configuration while Jiro is running.
25.4 Dynamic vs. Static Services

At the heart of all Jiro management solutions lies the management software. Management software, in the Jiro sense, consists of one or more services that provide customized management functionality to a network. These services—dynamic services—represent software that manages one or more resources.

Developers often find, when developing management solutions for any network, that it is necessary for their dynamic services to perform a fixed set of routine procedures. To aid dynamic-service developers and reduce development time, Jiro provides a set of standard services required by many management solutions. During initialization, Jiro starts a series of static services—also known as base services, because they commonly are used in distributed management solutions—and registers them with the Jini lookup service. The difference between dynamic services and static services is that dynamic services define the management solution that is deployed in a network and static services are tools that are globally available to all dynamic services to simplify development. The static services include:

1. the Jiro event service transmits events via the network to registered listeners;
2. the Jiro scheduling service fires events at scheduled times;
3. the Jiro log service enables logging of selected actions that occur in a Jiro network and any error messages that occur;
4. the Jiro transaction service is responsible for ensuring synchronized access to methods; and
5. the Jiro controller service creates controllers to enforce synchronized access to dynamic services.

Each Jiro domain has one instance of each standard service for use by remote clients or dynamic services. [Note: The transaction service and the controller service topics are beyond the scope of this chapter. For further information, please visit www.jiro.com.]

25.5 Dynamic Services

Jiro provides each management domain with a station that hosts dynamic services and enables remote clients to invoke methods on the hosted dynamic services. Jiro provides the station at startup.

In Jiro, a dynamic service that provides access to a managed resource or a set of related dynamic services is called a Management Facade. A Jiro component’s Management Facade enables other Jiro components to manage those devices to which the Management Facade provides access. The Facade provides access to manage these devices by serving as a unique entry point for all Jiro components to interact with that Facade’s dynamic services. The Management Facade shields clients from that Facade’s complex underlying structure and behavior.

The Management Facade is an example of the Facade design pattern. This design pattern reduces system complexity, because a client interacts with only one object—called the facade object. In the field of software development, the Facade design pattern shields applications developers from subsystem complexities. Developers need to be familiar with only the operations of the facade object, rather than with the more detailed operations of the entire subsystem.
When driving a car, you know that pressing the gas pedal accelerates your car, but you are unaware of exactly how the gas pedal causes your car to accelerate. The facade object provides a simple interface for the behaviors of a subsystem—an aggregate of objects that collectively comprise a major system responsibility. The gas pedal, for example, is the facade object for the car’s acceleration subsystem, the steering wheel is the facade object for the car’s steering subsystem and the brake is the facade object for the car’s deceleration subsystem. A client object uses the facade object to interact with the objects behind the facade. The client remains unaware of how the objects behind the facade fulfill responsibilities, so the subsystem complexity is hidden from the client. When you press the gas pedal you act as a client object. In Jiro, the Management Facade is the facade object, and Jiro components that use the Management Facade to manage underlying resources are the client objects.

To make a dynamic service available in a management domain, dynamic-service providers must follow two steps—implement the dynamic service (a step performed by a developer) and deploy the dynamic service to a station in a management domain (a step usually performed by a system administrator). In the rest of this section, we demonstrate how to develop, deploy and access a dynamic service that manages a network printer. As in Chapter 24, this chapter uses a printer simulator to represent the resource that our dynamic service manages.

The printer interface has a limited number of operations. Our dynamic service exposes this interface. We will create a management console that enables a system administrator to manage the printer remotely. The resulting management solution provides a simple example of a typical Jiro management solution.

### 25.5.1 Dynamic-Service Implementation

To implement a dynamic service, a developer must first define a public interface for the service. The public interface exposes to clients and to other dynamic services the operations one can perform on a resource that will be managed.

PrinterManagement (Fig. 25.4) exposes the management operations we can perform on the printer. From this interface, a developer will be able to provide access to the printer’s state at any given moment, through such methods as isPrinting, isOnline, isPaperJam and getPendingPrintJobs. Developers will also have the ability to operate the printer, through such operations as addPaper, addToner and cancelPendingPrintJobs. In our implementation, the PrinterManagement dynamic service will schedule the printer to shut down every weekend. Operation terminateScheduledTasks permits clients to cancel all tasks scheduled for the printer by the PrinterManagement dynamic service.

---

1 // Fig. 25.4: PrinterManagement.java
2 // This class defines the interface for the dynamic service.
3 package com.deitel.advjhtp1.jiro.DynamicService.service;
4

---

2. Resources in the Jiro sense are not limited to physical devices on a network. They also include other dynamic services that can be accessed through the Jiro station or through other means.
Class **PrinterManagementImpl** (Fig. 25.5) is the dynamic service implementation that we developed for the network printer simulator (class **Printer** in Fig. 25.8). Note that **PrinterManagementImpl** uses static services that we explain in subsequent sections (see Section 25.6 for more details).
import net.jini.core.event.*;
import net.jini.core.entry.*;
import net.jini.core.lease.*;

import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.entry.*;

import javax.fma.services.ServiceFinder;
import javax.fma.services.event.EventService;
import javax.fma.services.log.LogMessage;
import javax.fma.services.log.LogService;
import javax.fma.services.scheduling.SchedulingService;
import javax.fma.services.scheduling.SchedulingService.*;
import javax.fma.util.*;

import com.deitel.advjhtp1.jiro.DynamicService.printer.*;

public class PrinterManagementImpl 
        implements PrinterManagement {

    private Printer printer;
    private LogService logService = null;
    private Lease observerLease;
    private LeaseRenewalManager leaseRenewalManager;
    private Ticket turnOffPrinter;
    private Ticket turnOnPrinter;

    // default constructor
    public PrinterManagementImpl()
    {
        System.out.println( "Dynamic service started.\n" );

        // start a printer
        printer = new Printer();
        Thread printerThread = new Thread( printer );
        printerThread.start();

        // subscribe to printer events
        try {

            // get the event service
            EventService eventService = 
                    ServiceFinder.getEventService();

            // get the log service
            logService = ServiceFinder.getLogService();

            // other code...
// get a listener
PrintEventListener listener =
    new PrintEventListener( this );

// subscribe as an observing listener to event with
// topic ".Printer.Error"
observerLease = eventService.subscribeObserver(
    ".Printer.Error", listener, null, 10 * 60 * 1000 );

// renew observer listener’s lease
leaseRenewalManager = new LeaseRenewalManager();
leaseRenewalManager.renewUntil(
    observerLease, Lease.FOREVER, null );

// get the scheduling service
SchedulingService schedulingService =
    ServiceFinder.getSchedulingService();

// define the schedule for turn-off printer task
// printer is turned off at 8:00PM every friday
GregorianCalendar calendar = new GregorianCalendar();
calendar.set( 2001, 7, 27 );
Date startDate = calendar.getTime();
calendar.set( 2003, 7, 27 );
Date endDate = calendar.getTime();
int[] monthsOff = { Calendar.JANUARY,
    Calendar.FEBRUARY, Calendar.MARCH, Calendar.APRIL,
    Calendar.MAY, Calendar.JUNE, Calendar.JULY,
    Calendar.AUGUST, Calendar.SEPTEMBER,
    Calendar.OCTOBER, Calendar.NOVEMBER,
    Calendar.DECEMBER };;
int[] daysOfWeekOff = { Calendar.FRIDAY };;
int[] hoursOff = { 20 };
int[] minutesOff = { 0 };

// create the schedule for turn-off printer task
Schedule turnOffSchedule =
    schedulingService.newRepeatedDateSchedule(
        startDate, endDate, monthsOff, null,
        daysOfWeekOff, hoursOff, minutesOff,
        calendar.getTimeZone() );

// create the message description
LocalizableMessage turnOffMessage =
    new LocalizableMessage( PrinterManagementImpl.class,
        "TurnOffPrinter", null, null );

// define the handback object of the
// turn-off printer task
MarshalledObject handbackOff = new MarshalledObject(
    new String( "turn-off" ) );

Fig. 25.5 PrinterManagementImpl implementation of interface
PrinterManagement (part 3 of 8).
// schedule task and get the Ticket for
turnOffPrinter = schedulingService.scheduleTask(
    listener, turnOffMessage, turnOffSchedule,
    SchedulingService.NONE, handbackOff );

// define the schedule for turn-on printer task
// printer is turned on at 7:00AM every Monday
calendar = new GregorianCalendar();
calendar.set( 2001, 7, 27 );
startDate = calendar.getTime();
calendar.set( 2003, 7, 27 );
endDate = calendar.getTime();
int[] monthsOn = { Calendar.JANUARY,
    Calendar.FEBRUARY, Calendar.MARCH, Calendar.APRIL,
    Calendar.MAY, Calendar.JUNE, Calendar.JULY,
    Calendar.AUGUST, Calendar.SEPTEMBER,
    Calendar.OCTOBER, Calendar.NOVEMBER,
    Calendar.DECEMBER };
int[] daysOfWeekOn = { Calendar.MONDAY  };
int[] hoursOn = { 7 };
int[] minutesOn = { 0 };
Schedule turnOnSchedule =
    schedulingService.newRepeatedDateSchedule(
        startDate, endDate, monthsOn, null,
        daysOfWeekOn, hoursOn, minutesOn,
        calendar.getTimeZone() );

// create the message description
LocalizableMessage turnOnMessage =
    new LocalizableMessage( PrinterManagementImpl.class,
    "TurnOnPrinter", null, null );

// define the handback object of the
// turn-on printer task
MarshalledObject handbackOn = new MarshalledObject(
    new String( "turn-on" ) );

// schedule task and get the Ticket for
// the scheduled task
turnOnPrinter = schedulingService.scheduleTask(
    listener, turnOnMessage, turnOnSchedule,
    SchedulingService.NONE, handbackOn );

} // end try

// handle exception scheduling task
catch ( Exception exception ) {
    System.out.println( "PrinterManagementImpl: " +
    "Exception occurred when scheduling tasks." );
System.out.println( "Please read debug file ...
" );
            Debug.debugException( "scheduling task", exception );
        }
    } // end PrinterManagementImpl constructor

    // cancel scheduled tasks
    public void terminateScheduledTasks()
    {
        // cancel turn-on and turn-off printer tasks
        try {
            turnOffPrinter.cancel();
            turnOnPrinter.cancel();
        }

        // handle exception canceling scheduled task
        catch ( Exception exception ) {
            System.out.println( "PrinterManagementImpl: " +
                                "Exception occurred when canceling tasks." );
            System.out.println( "Please read debug file ...
" );
            Debug.debugException( "cancel scheduled task", exception );
        }
    } // end cancel scheduled task

    // add paper to printer
    public void addPaper( int amount )
    {
        System.out.println( "PrinterManagementImpl: Adding paper ...
" );
        printer.replenishPaperTray( amount );
    }

    // is printer printing?
    public boolean isPrinting()
    {
        return printer.isPrinting();
    }

    // is printer jammed?
    public boolean isPaperJam()
    {
        return printer.isPaperJam();
    }

    // get printer's pages count
    public int getPaperInTray()
    {
        return printer.getPaperInTray();
    }

    Fig. 25.5  PrinterManagementImpl implementation of interface PrinterManagement (part 5 of 8).
// get pending jobs
public String[] getPendingPrintJobs()
{
    return printer.getPendingPrintJobs();
}

// is printer online?
public boolean isOnline()
{
    return printer.isOnline();
}

// cancel pending printing jobs
public void cancelPendingPrintJobs()
{
    System.out.println( "PrinterManagementImpl: 
    + "Canceling pending print jobs ... \n" );
    printer.cancelPendingPrintJobs();
}

// receive notifications
public void notify( RemoteEvent remoteEvent )
throws UnknownEventException, RemoteException
{
    String subString =
    "com.deitel.advjhtp1.jiro.DynamicService.printer";
    String source = ( String ) remoteEvent.getSource();
    source = source.substring( 0, subString.length() );

    // printer event
    if ( source.equals( subString ) )
        eventHandler( remoteEvent );

    // scheduled task
    else
        performTask( remoteEvent );
}

// add toner to printer
public void addToner()
{
    System.out.println(
    "PrinterManagementImpl: Adding toner ...\n" );
    printer.addToner();
}

// perform task when scheduled time arrives
private void performTask( RemoteEvent remoteEvent )
{
    // perform task
    try {

Fig. 25.5 PrinterManagementImpl implementation of interface PrinterManagement (part 6 of 8).
// get task type
String type =
  ( String ) remoteEvent.getRegistrationObject().get();

// turn-off printer
if ( type.equals( "turn-off" ) )
  printer.setOffline();

// turn-on printer
else if ( type.equals( "turn-on" ) )
  printer.setOnline();

// handle exception performing scheduled task
  catch ( Exception exception ) {
    System.out.println( "PrinterManagementImpl: " +
      "Exception occurred when performing tasks." );
    System.out.println( "Please read debug file ...
    Debug.debugException( "perform scheduled task", exception );
  }

} // end method performTask

// handle event
private synchronized void eventHandler ( RemoteEvent remoteEvent )
{
  String source = ( String ) remoteEvent.getSource();

  // generate the log message
  Serializable params[] = new Serializable[ 2 ];
  params[ 0 ] = source;
  params[ 1 ] = new Date();

  // define localizable message
  LocalizableMessage localizableMessage =
    new LocalizableMessage( PrinterManagementImpl.class,
      "Event", params, Locale.US );

  // define log message
    + source, null );

  // post the log message
  try {
    logService.log( logMessage );
  }

Fig. 25.5 PrinterManagementImpl implementation of interface PrinterManagement (part 7 of 8).
Access to dynamic services occurs through proxy objects that a client instantiates. The *jirocw* tool generates a dynamic service’s proxy class at deployment time. We discuss the deployment of this management application in Section 25.7. We generate *PrinterManagementImplProxy*—the proxy class for dynamic service implementation *PrinterManagementImpl*—using the *jirocw* tool. The only means by which a dynamic service can be instantiated is through instantiating its proxy object. When a client instantiates *PrinterManagementImplProxy*, the proxy remotely calls the no-argument constructor of the *PrinterManagementImpl* dynamic service.

If the dynamic service implements method *getLookupEntries* (lines 334–343), the Jiro station submits a *PrinterManagementImplProxy* reference to the domain’s Jini lookup service with the *Entry* attributes returned by *getLookupEntries* at instantiation time. Dynamic services and clients wanting to interact with the *PrinterManagement* dynamic service after a different client instantiates the proxy, must do so through the *PrinterManagementImplProxy* reference stored in the Jini lookup service.

Lines 52–54 start the thread that will simulate the printer we are managing. Class *Printer*’s *run* method controls the thread. Lines 60–82 obtain references to several static services used by the *PrinterManagement* dynamic service. We mention each static service here and discuss each in detail in Section 25.6. The standard static services are obtained through helper class *ServiceFinder* (discussed in Section 25.6.1). Lines 60–61 obtain a reference to the *event service* (see Section 25.6.2). Line 64 obtains a reference to the *log service* (Section 25.6.3). Lines 67–68 create a new *PrinterEventListener* to listen for events originating from the *Printer*. Lines 72–73 subscribe *PrinterEventListener* in the event service. Lines 80–81 renew the lease granted to the listener. Lines 81-82 obtain a reference to the *scheduling service* (Section 25.6.4). Lines

```java
323  // handle exception posting log message
324  catch ( Exception exception ) {
325      System.out.println( "PrinterManagementImpl: " +
326          "Exception occurred when posting log message." );
327      System.out.println( "Please read debug file ...\n" );
328      Debug.debugException( "log service", exception );
329  }
330
331  } // end eventHandler
332
333  // entry object
334  private Entry[] getLookupEntries() {
335      return ( new Entry[] {
336          new ServiceInfo( "PrinterManagementImpl",
337              "Deitel Association, Inc.",
338              "Deitel Association, Inc",
339              "1.0", "Model 0", "0.0.0.1" )
340      } );
341  }
342
343  }
344
Fig. 25.5 *PrinterManagementImpl* implementation of interface *PrinterManagement* (part 8 of 8).
86–122 and 126–162 schedule two events—*turn printer off* and *turn printer on*—to occur every weekend.

Method **terminateScheduledTasks** (lines 177–194) will terminate all scheduled tasks. Method **addPaper** will add paper to the printer by delegating its call to **Printer**. Methods **isOnline**, **isPrinting**, **isPaperJam**, **getPaperTray**, **getPendingPrintJobs**, **addToner** and **cancelPendingPrintJobs** follow the pattern of **addPaper**, delegating calls to their respective methods in **Printer**.

Method **notify** (lines 243–257) receives delegated notifications from a **PrinterEventListener** and determines whether the event notification is an event triggered by **Printer** or by the scheduling service. Method **notify** calls to method **performTask** (lines 268–295) if the event received was triggered by the scheduling service. Lines 252–256 determine the type of event and perform the corresponding action. Method **notify** makes a call to method **eventHandler** (lines 298–331) if the event received was triggered by the **Printer**. Method **eventHandler** logs a message pertaining to the event received (lines 304–320).

Lines 334–343 implement method **getLookupEntries**—the only way to declare a class as a dynamic service. A class that presents a declaration of method **getLookupEntries** is called a *point object*. Dynamic services may involve more than one class to provide the underlying definition of a dynamic service. The Jiro specification states that only one object, the *entry point* for a dynamic service instance, may exist. Instantiating the entry-point object is the only allowed way to instantiate a dynamic service and the classes on which it depends. Method **getLookupEntries** returns a set of entries. These entries are used by the station to register the dynamic service’s proxy in the Jini lookup service of the domain.

Figure 25.6 (**PrinterEventListener.java**) is the event listener instantiated in **PrinterManagementImpl** lines 67–68. **PrinterEventListener** implements interface **RemoteEventListener**—Jiro requires this for all event subscribers. Figure 25.6 shows our **RemoteEventListener** implementation. The event service calls method **notify** (lines 36–41) when an event occurs.

```java
// PrinterEventListener.java
// This class defines the listener that listens for events
// issued by a printer.
package com.deitel.advjhtp1.jiro.DynamicService.service;

// Java core packages
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

// Jini core packages
import net.jini.core.event.*;

public class PrinterEventListener
    implements RemoteEventListener {
    private RemoteEventListener eventListener;
```

*Fig. 25.6* **PrinterEventListener** used by all classes subscribed for events from **Printer** (part 1 of 2).
25.6 Static Services

Printer (Fig. 25.8) and PrinterManagementImpl (Fig. 25.5) use Jiro’s static services. Printer publishes events to the event service (line 320 of Fig. 25.8). Class PrinterManagementImpl uses the event service to listen for events, the log service to record messages pertaining to events received and the scheduling service to schedule the printer to go offline for servicing every week from 8PM on Friday to 7AM on Monday. The following subsections explain how to obtain and use Jiro’s static services. The case study does not use all the static services provided by Jiro, so the following sections discuss the remaining services.

25.6.1 Locating Static Services with Class ServiceFinder

Before using the static services, we need to get the service proxies of these static services. Class ServiceFinder (package javax.fma.service) provides methods for locating the static services. The class contains ten methods—two for each static service. To get the proxy for a static service, an application or dynamic service either calls method ServiceFinder.getServiceName with no arguments, which locates the service for the local management domain, or calls method ServiceFinder.getServiceName with an argument that specifies the management domain, which locates the service for that specific management domain. For example, method getLogService with no arguments returns...
the log service for the local management domain. Similarly, static services can also be found through the Jini lookup service. **ServiceFinder** is a convenience class that encapsulates the calls to the Jini lookup service.

### 25.6.2 Event Service

*Event* objects encapsulate an action that happened in a network (locally or remotely). They allow actions that occur in one part of the network to be handled in another part. *Publish-subscribe* event systems, such as Jiro’s, are one type of event-driven network. Such networks generally are composed of three types of elements—the *event publisher* that propagates events throughout the network, the *event subscriber* that listens on one end of the network for events that are published and the events. The *event service* is a bridge between the event publisher and the event subscriber. In the context of Jiro, an event publisher posts an event or series of events to the event service. An event subscriber receives the event(s) from the event service. The event subscriber can subscribe to types of events—also known as *topics* of events—that have not yet been published.

An event subscriber can subscribe to the event service as an *observer listener* or a *responsible listener*. An observer listener can receive an event, but cannot handle or consume an event. A responsible listener has special priority to respond and consume specific events. Responsible listeners must decide whether or not to consume an event. If a responsible listener decides not to consume an event, its *notify* method must throw an **EventNotHandledException**. This causes the event service to propagate the event to the next responsible listener in the chain.

Jiro’s event service uses the *Chain-of-Responsibility design pattern* to ensure that a responsible listener handles an event. Often, a system needs to determine at run time the object that will handle a particular message. For example, consider the design of a three-line office phone system. When a person calls the office, the first line handles the call—if the first line is busy, the second line handles the call, and if the second line is busy, the third line handles the call. If all lines in the system are busy, an automated speaker instructs that person to wait for the next available line—when a line becomes available, that line handles the call. When an event occurs that applies to both responsible listeners and observer listeners, all observer listeners and the first responsible listener in the chain will get the event.

The Chain-of-Responsibility design pattern enables a system to determine at run time the object that will handle a message. This pattern allows an object to send a message to several objects in a *chain* of objects. Each object in the chain either may handle the message or pass the message to the next object in the chain. For instance, the first line in the phone system is the first object in the chain of responsibility, the second line is the second object, the third line is the third object, and the automated speaker is the fourth object. Note that this mechanism is not the final object in the chain—the next available line handles the message, and that line is the final object in the chain. The chain is created dynamically in response to the presence or absence of specific message handlers. In Jiro, the event publisher sends an event to a responsible listener—if the listener cannot handle the event, the Event Service sends the event to another responsible listener. The Event Service propagates this event to other listeners, until a listener handles the event.

---

3. Clients can find the local management domain information through the **javax.fma.domain** system property.
Class **PrinterManagementImpl** (Fig. 25.5) is an example of a class that subscribes a listener to the event service. Class **PrinterManagementImpl** registers its **PrinterEventListener** as an observer listener upon initialization. Lines 72–73 subscribe to events with "*.Printer.Error" as their topic. Method **subscribeObserver** takes four arguments—a **String** that specifies the topic of the event, a listener that implements the **RemoteEventListener** so that notifications can be sent when the event service receives events for that topic, a **MarshalledObject** that is passed to the listener with each event (may be **null**) and a **long** value that specifies the length of the lease for the listener’s subscription. In the Jiro architecture, topics classify events and enable the event service to deliver events to the proper subscriber(s). A topic is a dot-delimited **String** that defines a tree-like structure. This structure helps identify event types and related subtypes.

All event types start at the root—".". A simple dot represents the root of the tree-like structure. All elements belong to the root; therefore, all event topics match the "." string. ".A" represents all events of topic ".A". To define a subtype for this topic, append a period and the subtype name. Examples of two possible subtype topics for event topic ".A" could be ".A.aa" and ".A.bb". If an event subscriber registers with the event service as an observer for all events of topic ".A", the event subscriber will receive events of topic ".A" as well as events of topic ".A.aa" and ".A.bb". If a listener subscribes to event topics of type ".A.bb", only events of type ".A.bb" and its subtypes will be received.

Any event object posted to the event service must extend **Event** (package **javax.fma.services.event**). Class **PrinterErrorEvent** (Fig. 25.7) extends class **Event**. **Event** classes must implement method **clone**, so that the event service can copy event-objects and pass them to all the subscribed listeners (lines 18–21).

```
// PrinterErrorEvent.java
// This class defines the events issued by a printer.
package com.deitel.advjhtp1.jiro.DynamicService.printer;

// Jiro package
import javax.fma.services.event.Event;

public class PrinterErrorEvent
    extends Event implements Cloneable {

    // PrinterErrorEvent constructor
    public PrinterErrorEvent( Object source, String topic )
    {
        super ( source, topic );
    }

    // clone event
    public Object clone()
    {
        return new PrinterErrorEvent( source, getTopic() );
    }
}
```

**Fig. 25.7** Custom error class thrown by Printer.
Class **Printer** (Fig. 25.8) is a slightly different implementation from class **PrinterSimulator** implementation presented in Chapter 24. (See Chapter 24 for implementation details on class **Printer**). Class **Printer** is an example of a class that publishes events to the event service. Method **fireEvent** (lines 308–328) publishes all events originating from the printer. Lines 314–317 create a **PrinterErrorEvent** (Fig. 25.7) object and initialize it to the proper value. The first argument in the **PrinterErrorEvent** is the source. We have chosen our source to be a **String** that describes where the event originated, plus additional information pertaining to the type of event that occurred. The second argument contains the event topic. The topic’s root is ".**Printer.Error"; the child of the event can be either **OutofPaper**, **LowToner** or **PaperJam**. A source of an event can be any object except **null**. To post an event, call method **post** (line 320). Method **post** takes one argument—the event to post for delivery to the listeners.

```java
// Printer.java
// This class simulates a printer device on a network.
// deitel package
package com.deitel.advjhtp1.jiro.DynamicService.printer;

// java core package
import java.util.Stack;
import java.rmi.*;
import java.io.*;

// Jiro packages
import javax.fma.services.ServiceFinder;
import javax.fma.services.event.EventService;
import javax.fma.util.*;

public class Printer implements Runnable {

    private Stack printerStack = new Stack();
    private boolean isPrinting = false;
    private boolean isPaperJam = false;
    private boolean isOnline = true;

    // 50 sheets of paper in tray
    private int paperInTray = 50;

    // 100% full of ink
    private int tonerCartridge = 100;

    private String currentPrintJob;
    private boolean isAlive = true;

    private EventService eventService;

    // printer constructor
    public Printer()
    {

Fig. 25.8 Printer simulator implementation (part 1 of 7).
```
// get EventService at gaven management domain
try {
    eventService = ServiceFinder.getEventService();
}

// handle exception getting EventService
catch ( Exception exception ) {
    Debug.debugException("getting EventService", exception);
}

// stops execution of thread
public void stop() {
    isAlive = false;
}

// main life-cycle of the printer.
// prints one job from print job stack
// 1) if offline, it pauses and waits.
// 2) if online, handles one print job
public void run() {
    // main loop within thread
    while ( isAlive ) {
        // printer will be offline
        if ( !isOnline ) {
            synchronized ( this ) {
                // waits for printer become online
                try {
                    wait();
                }
                // handle exception waiting
                catch ( InterruptedException exception ) {
                    Debug.debugException("printer wait", exception);
                }
            } // end synchronized
        } // end if
        // prints one job from print job stack
        startPrintingProcess();
    } // end while
} // end run
private synchronized void startPrintingProcess()
{
    // warm up the printer, print top print job from print
    // stack and adjust paper values and toner values
    try {
        // warm up printer for incoming batch of print jobs
        Thread.sleep( 1000 * 2 );

        if ( isOnline && ( paperInTray > 0 ) &&
            ( tonerCartridge > 10 ) && ( !isPaperJam ) ) {

            // start the printing process
            currentPrintJob = getNextPrintJob();
            isPrinting = true;

            // 12 seconds to print a normal document
            Thread.sleep( 1000 * 12 );

            // each print job uses 10 pages
            updatePaperInTray( paperInTray - 10 );
            updateToner();
            updatePaperJam();
            isPrinting = false;

            // make sure no references are left dangling
            currentPrintJob = null;

        } // end if
    }

    // handle exception starting printing process
    catch( InterruptedException exception ) {
        Debug.debugException(
            "starting printing process", exception );
    }
} // end method startPrintingProcess

// returns current printed job
private String getCurrentPrintJob()
{
    return currentPrintJob;
}

// update amount of paper in paper tray
private synchronized void updatePaperInTray( int newValue )
{
    paperInTray = newValue;
}
// fire event if paper tray low
if ( paperInTray <= 0 ) {
    System.out.println( "Printer: out of paper. " );
    fireEvent( "OutofPaper" );
}

// is paper jammed?
public boolean isPaperJam()
{
    return isPaperJam;
}

// is printer printing?
public boolean isPrinting()
{
    return isPrinting;
}

// is printer online?
public boolean isOnline()
{
    return isOnline;
}

// return number of pages in paper tray
public synchronized int getPaperInTray()
{
    return paperInTray;
}

// update amount of toner available in toner cartridge
public synchronized void updateToner()
{
    // after every print job, toner levels drop 1%
    tonerCartridge = tonerCartridge - 1;

    // fire event if toner is low
    if ( tonerCartridge <= 10 ) {
        System.out.println( "Printer: low toner. " );
        fireEvent( "LowToner" );
    }
}

// update paper jam
public synchronized void updatePaperJam()
{
    if ( Math.random() > 0.9 ) {
        isPaperJam = true;
        System.out.println( "Printer: paper jam. " );
        fireEvent( "PaperJam" );
    }
}
194 // return amount of toner in toner cartridge
195 public synchronized int getToner()
196 {
197     return tonerCartridge;
198 }
199
200 // replenish amount of paper in paper tray to specified value
201 public void replenishPaperTray ( int paperStack )
202 {
203     System.out.println( "Printer: adding " + paperStack
204             + " pages to printer ... \n" );
205     updatePaperInTray ( paperInTray + paperStack ) ;
206 }
207
208 // generates a random number of print jobs with varying IDs
209 private synchronized void populatePrintStack()
210 {
211     int numOfJobs = ( int ) ( Math.random ( ) * 10 ) + 1;
212     // generate print jobs
213     for ( int i = 0; i < numOfJobs ; i++ ) {
214         synchronized ( printerStack ) {
215             printerStack.add ( "PRINT_JOB_ID #" + i );
216         }
217     }
218     // add toner
219     public synchronized void addToner()
220     {
221         System.out.println( "Printer: adding toner . . . \n" );
222         tonerCartridge = 100;
223     }
224     // cancel pending print jobs
225     public synchronized void cancelPendingPrintJobs()
226     {
227         synchronized ( printerStack ) {
228             printerStack.clear();
229         }
230     }
231     // return next print job in stack, populating the stack if it is empty
232     private synchronized String getNextPrintJob()
233     {
234         if ( printerStack.isEmpty() ) {
235             populatePrintStack ( );
236         }
237     
238     }
239 }
240
241
242
Fig. 25.8 Printer simulator implementation (part 5 of 7).
// simulates absence of print jobs
try {
    Thread.sleep (
        ( int ) ( Math.random() * 1000 * 10 ) );
} 

// handle exception thread sleep
catch ( InterruptedException exception ) {
    Debug.debugException(
        "getting next print job", exception );
}

// Remove topmost queued resource.
String nextJob;
synchronized ( printerStack ) {
    nextJob = ( String ) printerStack.pop();
}
return nextJob;

} // end method getNextPrintJob

// return all jobs yet to be printed
public synchronized String[] getPendingPrintJobs() {
    String[] pendingJobs;
synchronized ( printerStack ) {
        Object[] temp = printerStack.toArray() ;
        pendingJobs = new String[ temp.length ] ;
    for ( int i = 0; i < pendingJobs.length ; i++ ) {
          pendingJobs [ i ] = ( String ) temp[ i ];
    }
    return pendingJobs;
}

// set printer status to online
public void setOnline() {
    System.out.println( "Printer: setting online ... \n" );
isOnline = true;
synchronized ( this ) {
    notifyAll();
}

Fig. 25.8 Printer simulator implementation (part 6 of 7).
25.6.3 Log Service

The log service can be to log information when an important event such as application invocation or an error occurs. The log service provides methods log and search to access to the log repository. Each message written to the log repository must be in the form of a javax.fma.service.log.LogMessage object. The log service is internationalized. This means that applications can adapt logged messages for viewing in different languages and different regions. The client or the service can either post a log message to the log service or retrieve log messages from the log service, obeying search criteria.

Class PrinterManagementImpl (Fig. 25.5) uses the log service to log every event it receives from the printer. This class uses ServiceFinder to get the service proxy for the log service. Line 64 of PrinterManagementImpl gets the log service by calling method ServiceFinder.getLogService with no arguments.

A log message contains a localizable message (a message in a format that any application can convert to a readable format for a given locale), a category to specify the type of log message and a Throwable—posted if the message is created when an error condition arises. The category of a log message must be a dot-delimited String that begins with
one of the following standard categories: LogMessage.AUDIT, LogMessage.DEBUG, LogMessage.WARNING, LogMessage.INFO, LogMessage.ERROR, LogMessage.TRACE. Lines 309–311 of PrinterManagementImpl create a localizable message. The LocalizableMessage (package javax.fma.util) constructor takes four arguments—a Class representing the localization resource file, a String that specifies the message key, an array of Serializable objects and a Locale that is used to create the fall-back text for this message. Both the Class and the String arguments passed to the LocalizableMessage cannot be null, otherwise an IllegalArgumentException will be thrown. Both the Serializable array and the Locale may be null. Both serve special purposes that are out of the scope of this example—the Serializable array serves as a way of storing object types within the message, the Locale helps specify what language format the message is originally stored in. If a client cannot convert the LocalizableMessage to its current locale, then a the log service generates a fall-back message, using the Locale. If a client passes null as the Locale argument, the default locale is used. In this example, the class specifies the PrinterManagementImpl.class as the root of the resource file and "TurnOnPrinter" as the message key. A properties file that will be associated to LocalizableMessage must be created and be placed in a specified directory. Instructions on how to create the properties file will be provided shortly. The properties file contain a "key=value" pair with "TurnOnPrinter" as the message key.

The log message is created in lines 314–316 of PrinterManagementImpl. The LogMessage (package javax.fma.service.log) constructor takes three arguments—a LocalizableMessage that contains the message, a String that indicates the category of the message and a Throwable that contains the stored exceptions. Both the LocalizableMessage argument and the category String arguments cannot be null. The Throwable argument can be null. The class specifies the category of the log message as TRACE.printer.source, where source is the event source. Line 320 of PrinterManagementImpl posts the log message to the log service by calling method log of interface LogService.

Before executing an application that uses the log service, you must to provide a resource file. The resource file for PrinterManagementImpl must have the name PrinterManagementImpl.properties. This file should be stored in the resources directory where PrinterManagementImpl.class is located. In our example, PrinterManagementImpl.class is located at

```
C:\com\deitel\advjhtp1\jiro\DynamicService\service
```

so the PrinterManagementImpl.properties resource file should be located at

```
C:\com\deitel\advjhtp1\jiro\DynamicService\service\resources
```

The resource file contains messageKey=messageText pairs. Both the message key and the message text must be strings. Figure 25.9 shows the resource file used in PrinterManagement application.

---

4. Contents of the serializable array will replace the numbers enclosed within braces (e.g., {0}, {1}, {2}) in a log message template obtained from the properties file.
The Jiro SDK has a `viewlog` tool to view log messages. To start the `viewlog` tool, open a command prompt and type

```
viewlog -domain jiro:JIROTEST
```

This command starts the `viewlog` tool, which displays log messages for domain `jiro:JIROTEST`. Figure 25.32 shows the `viewlog` tool GUI. For information about how to use this tool, refer to the *Jiro Technology SDK programmer’s Reference* in the Jiro SDK installation directory’s `docs` subdirectory. The Jiro Technology SDK Programmer’s Reference is available in PDF (`install_config.pdf`) and postscript (`install_config.ps`) formats.

### Common Programming Error 25.1

Class `LocalizableMessage` throws a `javax.fma.util.LocalizableMessage.LocalizationError` if the resource file does not exist, the location of the resource file is not correct or the resource file does not contain the message key.

### Software Engineering Observation 25.3

Starting Jiro with the **Clean before start** checkbox checked clears the log messages.

#### 25.6.4 Scheduling Service

The scheduling service schedules tasks that to be performed at future times. The scheduling service provides methods to schedule tasks at specific times, repeatedly for a specific duration or repeatedly according to calendar dates.

Class `PrinterManagementImpl` (Fig. 25.5) uses the scheduling service to schedule the shutdown of the printer during the week, from Friday at 8PM to Monday at 7AM. Lines 81–82 obtain the scheduling service. All schedules are created by using a factory method that the scheduling service provides.

Lines 86–162 create a schedule for two tasks that will be performed based on a calendar date. `PrinterManagement` performs two tasks: shut down printer on Friday at 8PM and turn on printer on Monday at 7AM. The tasks are performed by two separate scheduled events. In both cases, method `newRepeatedDateSchedule` schedules the events. Method `newRepeatedDateSchedule` takes eight arguments—a `Date` that specifies the date and time on which the task will first be performed; a `Date` that specifies the date and time after which no task will be performed; an `int` array that specifies the months (0–11) during which the task will be performed; an `int` array that specifies the days in a month on which the task will be performed; an `int` array that specifies the days in a week on which the task will be performed; an `int` array that specifies the hours (0–23) at which the task will be performed; an `int` array that specifies the minutes (0–59) at which the task will be performed; and a `TimeZone` that specifies the time zone in which the task will be performed. If any of the arguments are `null`, an `IllegalArgumentException` occurs.
Method `newRepeatedDateSchedule` returns a `Schedule` object. Lines 102–106 and 160–162 of `PrinterManagementImpl` create the message descriptions for each schedule created. These descriptions are passed to method `scheduleTask` to schedule the task. Method `scheduleTask` takes five arguments—a task that implements `RemoteEventListener`, a `LocalizableMessage` that describes the task, a `Schedule` on which the task runs, an `int` that states the late-performance policy and a `MarshalledObject` that the scheduling service passes to the task listener when the scheduled event is fired. In our case, the `MarshalledObject` will be a `String` object that describes what task the dynamic service should perform. The task, the description of the task and the schedule of the task cannot be `null`, otherwise an `IllegalArgumentException` occurs. The late-performance policy specifies those tasks that must be performed upon restarting if a scheduling service misses scheduled tasks, such as during system shut down or for any reason that renders the service inoperable. There are three choices for the late performance policy—`SchedulingService.NONE` indicates no late performances, `SchedulingService.ONE` indicates one late performance and `SchedulingService.ALL` indicates the maximum integer value of `Integer` worth of late performances. If the policy is set to `NONE`, then late performances are not allowed. If the policy is set to `ONE`, then only one late performance is allowed. If the policy is set to `ALL`, then all late performances are allowed. Method `scheduleTask` returns a `Ticket` object that can be used later to cancel the task.

A listener that implements the `RemoteEventListener` interface is required to schedule tasks. At the scheduled times, the scheduling service calls method `notify` of classes implementing interface `RemoteEventListener`. The implementation of method `notify` performs or delegates the tasks received. Class `PrinterEventListener` (Fig. 25.6) is the `RemoteEventListener` in this example.

### 25.7 Dynamic Service Deployment

Various command-line tools provided by Jiro facilitate the complicated task of dynamic service deployment.

The first step in deploying our dynamic service is to generate a proxy for our dynamic services. Generating a proxy creates the entry-point object that is remotely accessible to clients. The proxy also is responsible for instantiating the dynamic service each time the proxy constructor is called. Jiro provides two tools to generate the proxy for a dynamic service—`jiroc` and `jirocw` (available in the `bin` directory of the Jiro SDK installation). The `jiroc` tool is a GUI tool to generate the proxy, while the `jirocw` tool is a command-line tool to generate the proxy. These tools are available in the `bin` directory of the Jiro SDK installation. To generate the proxy with `jirocw`, first compile the interface and implementation of the service (`PrinterManagement.java` and `PrinterManagementImpl.java`). These examples are compiled under Windows 2000. For instructions on how to compile under different operating systems, please refer to Java documentation. We assume that the JiroSDK was installed at the location `c:\jirosdk`.

You will need to set a `JIRO_CLASSPATH` environment variable. This variable contains the path to all necessary `.JAR` files located in the `JiroSDK\lib` directory—`jini-core.jar`, `jini-ext.jar`, `jiro.jar`, and `jiro-tools.jar`. For the Windows 2000 environment, the following command line sets the `JIRO_CLASSPATH`:
set JIRO_CLASSPATH=
c:\jirosdk\lib\jini-core.jar;
c:\jirosdk\lib\jini-ext.jar;
c:\jirosdk\lib\jiro.jar;
c:\jirosdk\lib\jiro-tools.jar

[Note: You can also set this environment variable permanently, so you do not need to type the preceding command in each new command window.] The following command lines compile the necessary files for our case study:

```
javac -classpath .;%JIRO_CLASSPATH% com\deitel\advjhtp1\jiro\DynamicService\common\*.java
javac -classpath .;%JIRO_CLASSPATH% com\deitel\advjhtp1\jiro\DynamicService\printer\*.java
javac -classpath .;%JIRO_CLASSPATH% com\deitel\advjhtp1\jiro\DynamicService\service\*.java
```

PrinterManagement.class should now exist. Next, use command jirocw to generate the dynamic-service entry-point-object proxy file5:

```
jirocw com.deitel.advjhtp1.jiro.DynamicService.service.PrinterManagementImpl
```

The resulting file (PrinterManagementProxy.java) is located in the com\deitel\advjhtp1\jiro\DynamicService\service directory. Next, compile the newly generated proxy file with the command:

```
javac -classpath .;%JIRO_CLASSPATH% com\deitel\advjhtp1\jiro\DynamicService\service\PrinterManagementImplProxy.java
```

PrinterManagement subscribes PrinterEventListener as an observer listener, therefore it must implement RemoteEventListener which uses RMI to deliver events. Generate the proxy stub for RemoteEventListener with the following command line:

```
rmic -classpath .;%JIRO_CLASSPATH% com.deitel.advjhtp1.jiro.DynamicService.service.PrinterEventListener
```

The next step involves creating a deployment JAR file that will contain all of our dynamic service’s class files, plus all related properties files. The following command line generates a JAR file called PrinterManagementService.jar:

```
jar -cvf PrinterManagementService.jar com\deitel\advjhtp1\jiro\DynamicService\common\*.class com\deitel\advjhtp1\jiro\DynamicService\printer\*.class com\deitel\advjhtp1\jiro\DynamicService\service\*.class com\deitel\advjhtp1\jiro\DynamicService\service\resources\*.properties
```

5. The root directory (in this case c:\) must be in CLASSPATH, because jirocw does not include a classpath parameter option.
We must create an interface JAR file. The interface JAR file contains all the interfaces that a dynamic service implements plus the classes these interfaces depend on:

```
jar -cvf PrinterManagementService-ifc.jar
    com\deitel\advjhtp1\jiro\DynamicService\service\PrinterManagement.class
    com\deitel\advjhtp1\jiro\DynamicService\service\PrinterManagementImplProxy.class
```

We also must create a download JAR file. The download JAR file contains all the classes that the clients of the dynamic service might need to load dynamically:

```
jar -cvf PrinterManagementService-dl.jar
    com\deitel\advjhtp1\jiro\DynamicService\service\PrinterManagement.class
    com\deitel\advjhtp1\jiro\DynamicService\service\PrinterManagementImplProxy.class
```

Finally, we must create an implementation JAR file. The implementation JAR file contains all the classes required to run the dynamic service.

```
jar -cvf PrinterManagementService-impl.jar
    com\deitel\advjhtp1\jiro\DynamicService\common\*.class
    com\deitel\advjhtp1\jiro\DynamicService\printer\*.class
    com\deitel\advjhtp1\jiro\DynamicService\service\*.class
    com\deitel\advjhtp1\jiro\DynamicService\service\resources\*.properties
```

At this point, all the files required by the jarpackw tool have been created. The jarpackw tool creates the JAR files we need to deploy dynamic services. To use the jarpackw tool, type

```
jarpackw -pool %JIRO_CLASSPATH%;
    c:\PrinterManagementService.jar
    -ifc PrinterManagementService-ifc.jar
    -impl PrinterManagementService-impl.jar
    -dl PrinterManagementService-dl.jar
```

The -pool option specifies the path to the source and the JAR files needed to execute the dynamic service. The -ifc option specifies the interface JAR file. The -impl option specifies the implementation JAR file. The -dl option specifies the download JAR file.

Finally, jardeploy tool deploys our dynamic service in the specified domain:

```
jardeploy -station SharedJiroStation -domain domainName
    -impl PrinterManagementService-impl.jar
    -dl PrinterManagementService-dl.jar -verbose -inventory
```

where domainName is the domain name in which to deploy our dynamic service. The command-line option station specifies the deployment station name. In this case, SharedJiroStation is the default name given to the Jiro station during installation. The domain option specifies the management domain on which the station is running. The impl option specifies the implementation JAR file. The dl option specifies the download JAR file. The verbose option activates the verbose mode to display status information
on the deployment of the dynamic service. The inventory option prints out the inventory of the station that contains deployed dynamic services.

If deployment is successful, the jardeploy tool displays the deployment results in the Output panel of the Igniter GUI. Figure 25.10 shows the deployment results.

**Common Programming Error 25.2**

A DeploymentException with the key "no_point_objects" occurs if an entry-point class is not found in the deployed JAR.

### 25.7.1 Dynamic-Service Usage

Deployment does not instantiate our dynamic service. We must do this by constructing an instance of the proxy we generated using the jirocw tool. **PrinterManagement-Starter** (Fig. 25.11) is a class that remotely instantiates the **PrinterManagement-Impl** dynamic service.

A **StationAddress** identifies the station to which we will connect. The **StationAddress** constructor requires eight arguments—a **String** that specifies the management domain on which the station runs, a **String** that specifies the role of the station, and six other fields inherited from the **ServiceInfo** entry. Line 35 calls **PrinterManagementImplProxy** constructor. Upon initialization, the Jiro station will register the proxy stub of **PrinterManagementImpl** in the Jini lookup service. From this point forward, all clients use this stub located in the lookup service to access the **PrinterManagementImpl** instance. Compile class **PrinterManagementStarter** and execute it to instantiate the **PrinterManagement** dynamic service.

---

---

---

---

---

Fig. 25.10 Deployment results.

---

6. **StationAddress**' fields are used to match Station Proxies registered in the Jini lookup service. All non null fields are matched exactly. Fields that are null act as wild cards, thus matching any values. Precisely specifying as many fields as possible is the key to accurate results.
// Fig. 25.11 : PrinterManagementStarter.java
// This application demonstrates how to obtain the proxies of
// a dynamic service.
package com.deitel.advjhtp1.jiro.DynamicService.client;

// Java core package
import java.rmi.*;

// Jiro packages
import javax.fma.common.*;

// Deitel packages
import com.deitel.advjhtp1.jiro.DynamicService.service.*;

public class PrinterManagementStarter {

    // PrinterManagementStarter constructor
    public PrinterManagementStarter( String domain ) {

        PrinterManagement managementProxy;

        // set security manager
        if ( System.getSecurityManager() == null )
            System.setSecurityManager( new RMISecurityManager() );

        // obtain station address
        StationAddress stationAddress =
            new StationAddress( domain,
                null, null, null, null, null, null, null );

        // get the proxies of dynamic services
        // and start the services
        try {
            managementProxy =
                new PrinterManagementImplProxy( stationAddress );
        }

        // handle exception getting proxies and starting policies
        catch ( RemoteException exception ) {
            exception.printStackTrace();
        }

    } // end PrinterManagementStarter constructor

    // method main
    public static void main( String args[] )
    {
        String domain = "";
    
}
After instantiating the `PrinterManagement` dynamic service, a reference to the dynamic service can be obtained through the Jini lookup service. We provide class `DynamicServiceFinder` (Fig. 25.12) as a helper class that encapsulates the Jini lookup service discovery process and the retrieval of services. For details on the Jini lookup service, Chapter 22.

Fig. 25.11 `PrinterManagementStarter` dynamic service instantiator program (part 2 of 2).

After instantiating the `PrinterManagement` dynamic service, a reference to the dynamic service can be obtained through the Jini lookup service. We provide class `DynamicServiceFinder` (Fig. 25.12) as a helper class that encapsulates the Jini lookup service discovery process and the retrieval of services. For details on the Jini lookup service, Chapter 22.

Fig. 25.12 Finds dynamic service proxies within a lookup service (part 1 of 3).
// DynamicServiceFinder constructor
public DynamicServiceFinder ( String domain, Entry[] serviceEntries )
{
    System.setSecurityManager( new RMISecurityManager() );
    entries = serviceEntries;
    LookupDiscovery lookupDiscovery = null;

    // discover lookup services
    try {
        lookupDiscovery = new LookupDiscovery(
            new String[] { domain } );
    }
    catch ( IOException exception ) {
        Debug.debugException(
            "discover lookup service", exception );
    }

    // catch the IOException
    catch ( IOException exception ) {
        Debug.debugException(
            "discover lookup service", exception );
    }

    // install a listener
    lookupDiscovery.addDiscoveryListener ( this );

    // wait until woken up by notification
    try {
        synchronized ( this ) {
            wait();
        }
    }
    catch ( Exception exception ) {
        Debug.debugException(
            "wait for lookup service", exception );
    }

    } // end DynamicServiceFinder constructor

// discover new lookup services
public void discovered( DiscoveryEvent event )
{
    // get the proxy registrars for those services
    registrars = event.getRegistrars();

    // wake up all waiting threads
    synchronized ( this ) {
        notifyAll();
    }
}
The `DynamicServiceFinder` constructor (lines 29–66) uses multicast request protocol to discover all Jini lookup services in the network. The constructor blocks until the first Jini lookup service is discovered. The constructor achieves the blocking behavior by calling method `wait` (line 56). Method `discovered` (lines 69–78) is called when a lookup service is discovered. This calls method `notifyAll` (line 76) to allow the constructor to continue execution. Method `getService` (lines 84–102) returns all matching entries from the first lookup service found.

Class `PrinterClientGUI` (Fig. 25.13) demonstrates remote method calls to the dynamic service. Our constructor (lines 48–160) takes one argument that specifies the management domain. Lines 271–274 register `RemoteEventListenerImpl` as an observer listener with the event service. Lines 275–276 create a lease renewal manager that renews the lease granted to the listener. `RemoteEventListenerImpl` receives all notifications of topic `.Printer.Error`. Note that `RemoteEventListenerImpl` is a convenience class provided by Jiro. It is a simple remote event listener implementation available to both clients and Jiro stations. This eliminates the need for dynamic class loading. Lines 66–75 define the upper panel (the printer status panel) of the GUI. Lines 78–110 define the middle panel (the button panel) of the GUI, which contains two buttons—`Check Status` and `Cancel Pending Print Jobs`. Lines 113–122 define the lower panel (the event panel) of the GUI, which displays the printer events. Lines 135–158 define a window listener for the output window. When the window is closed, the observer listener is released and the scheduled task is cancelled.

7. Please note that in our example, line 90 makes the assumption that only one Jini lookup service exists in the network. In some cases, multiple Jini lookup services may exist.
Method `checkStatusButtonAction` (lines 163–226) displays the printer’s status to the status panel when `Check Status` button is clicked. Lines 175–188 call the proxy’s methods to get the current status of the printer. Lines 197–224 process the information obtained and display it in the status panel.

Method `cancelJobsButtonAction` (lines 229–241) executes when the user clicks the `Cancel Pending Print Jobs` button. Line 233 invokes the proxy’s `cancelPendingPrintJobs` method to cancel pending print jobs.

Method `getPrinterManagementProxy` (lines 244–260) obtains a proxy reference to the `PrinterManagement` dynamic service. This reference enables us to invoke methods on the `PrinterManagement` dynamic service.

```java
// Fig. 25.13: PrinterClientGUI.java
// This application demonstrates how to obtain the proxy of
// a dynamic service and management policies, and call methods
// of the dynamic service.
package com.deitel.advjhtp1.jiro.DynamicService.client;

// Java core packages
import java.rmi.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;

// Java standard extensions
import javax.swing.*;

// Jini core package
import net.jini.core.lease.Lease;
import net.jini.core.event.*;
import net.jini.core.entry.Entry;

// Jini extension package
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.entry.Entry;

// Jiro packages
import javax.fma.common.*;
import javax.fma.services.*;
import javax.fma.services.event.EventService;
import com.sun.jiro.util.*;

// Deitel packages
import com.deitel.advjhtp1.jiro.DynamicService.common.*;
import com.deitel.advjhtp1.jiro.DynamicService.service.*;

public class PrinterClientGUI extends JFrame
    implements RemoteEventListener {
    private PrinterManagement printerManagementProxy;
```

**Fig. 25.13** Management console user interface (part 1 of 7).
private JTextArea printerStatusTextArea =
    new JTextArea();
private JTextArea printerEventTextArea =
    new JTextArea();
private Lease observerLease;
private LeaseRenewalManager leaseRenewalManager;

public PrinterClientGUI( String domain )
{
    super( "JIRO Printer Management Example" );

    // set security manager
    if ( System.getSecurityManager() == null )
        System.setSecurityManager( new RMISecurityManager() );

    // obtain reference to proxy
    printerManagementProxy =
        getPrinterManagementProxy( domain );

    // subscribe to event service as an observer listener
    subscriberObserver( domain );

    Container container = getContentPane();

    // status panel
    JPanel printerStatusPanel = new JPanel();
    printerStatusPanel.setPreferredSize(
        new Dimension( 512, 200 ) );
    JScrollPane statusScrollPane = new JScrollPane();
    statusScrollPane.setAutoscrolls( true );
    statusScrollPane.setPreferredSize(
        new Dimension( 400, 150 ) );
    statusScrollPane.getViewport().add(
        printerStatusTextArea, null );
    printerStatusPanel.add( statusScrollPane, null );

    // buttons panel
    JPanel buttonPanel = new JPanel();
    buttonPanel.setPreferredSize(
        new Dimension( 512, 200 ) );

    // define action for Check Status button
    JButton checkStatusButton =
        new JButton( "Check Status" );
    checkStatusButton.addActionListener(
        new ActionListener() {
            public void actionPerformed( ActionEvent event ) {
                checkStatusButtonAction( event );
            }
        });
// define action for Remove Jammed Paper button
JButton cancelJobsButton = new JButton("Cancel Pending Print Jobs");
cancelJobsButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        cancelJobsButtonAction(event);
    }
});

// add three buttons to the panel
buttonPanel.add(checkStatusButton, null);
buttonPanel.add(cancelJobsButton, null);

// events panel
JPanel printerEventPanel = new JPanel();
printerEventPanel.setPreferredSize(new Dimension(512, 200));
JScrollPane eventsScrollPane = new JScrollPane();
eventsScrollPane.setAutoscrolls(true);
eventsScrollPane.setPreferredSize(new Dimension(400, 150));
eventsScrollPane.getViewport().add(printerEventTextArea, null);
printerEventPanel.add(eventsScrollPane, null);

// initialize the text
printerStatusTextArea.setText("Printer Status: ---
");
printerEventTextArea.setText("Events: ---
");

// put all the panels together
container.add(printerStatusPanel, BorderLayout.NORTH);
container.add(printerEventPanel, BorderLayout.SOUTH);
container.add(buttonPanel, BorderLayout.CENTER);

// release observer listener and cancel scheduled task
// when window closing
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent event) {
        try {
            leaseRenewalManager.remove(observerLease);
        }
        catch (Exception e) {
            // error handling
        }
    }
});
// handle exception releasing observer listener
catch ( Exception exception ) {
    exception.printStackTrace();
}

// terminate the program
System.exit( 0 );

} // end method windowClosing

} // end WindowAdapter constructor

}; // end addWindowListener

} // end PrinterManagementGUI constructor

// check print status
public void checkStatusButtonAction( ActionEvent event )
{
    boolean isOnline = false;
    boolean isPaperJam = false;
    boolean isPrinting = false;
    int paperRemaining = 0;
    String[] pendingJobs = null;

    // manage printer remotely
    try {
        // check if the printer is on line
        isOnline = printerManagementProxy.isOnline();

        // check if the printer is paper jammed
        isPaperJam = printerManagementProxy.isPaperJam();

        // check if the printing is printing
        isPrinting = printerManagementProxy.isPrinting();

        // get the paper tray
        paperRemaining = printerManagementProxy.getPaperInTray();

        // get pending jobs
        pendingJobs =
            printerManagementProxy.getPendingPrintJobs();
    }

    // handle exception calling dynamic service methods
    catch ( Exception exception ) {
        exception.printStackTrace();
    }

    Fig. 25.13 Management console user interface (part 4 of 7).
// status for the online condition
if ( isOnline )
    printerStatusTextArea.append("\nPrinter is ONLINE.\n");
else
    printerStatusTextArea.append("\nPrinter is OFFLINE.\n");

// status for the paper jam condition
if ( isPaperJam )
    printerStatusTextArea.append("Paper jammed.\n");
else
    printerStatusTextArea.append("No Paper Jam.\n");

// status for the printing condition
if ( isPrinting )
    printerStatusTextArea.append("Printer is currently printing.\n");
else
    printerStatusTextArea.append("Printer is not printing.\n");

// status for paper tray condition
printerStatusTextArea.append("Printer paper tray has \n" + paperRemaining + " pages remaining.\n");

// number of pending jobs
printerStatusTextArea.append("Number of pending jobs: \n" + pendingJobs.length + "\n");

} // end method checkStatusButtonAction

// cancel print jobs
public void cancelJobsButtonAction( ActionEvent event )
{
    // cancel pending print jobs
    try {
        printerManagementProxy.cancelPendingPrintJobs();
    }

    // handle exception canceling pending print jobs
    catch ( Exception exception) {
        exception.printStackTrace();
    }
}

} // end method cancelJobsButtonAction

// get dynamic service proxy
public PrinterManagement getPrinterManagementProxy(
    String domain )
{

Entries entries = new Entry[] {
    new ServiceInfo( "PrinterManagementImpl",
        "Deitel Association, Inc.",
        "Deitel Association, Inc",
        "1.0", "Model 0", "0.0.0.1" )
};

DynamicServiceFinder finder = new DynamicServiceFinder( domain, entries );

// return dynamic service proxy
return ( PrinterManagement ) finder.getService();

} // end method getPrinterManagementProxy

// listener for events
public void subscriberObserver( String domain ) {
    // subscribe to printer events
    try {
        EventService eventService =
            ServiceFinder.getEventService( domain );

        // subscribe as an observing listener to certain event
        RemoteEventListener listener =
            new RemoteEventListenerImpl( this );
        observerLease = eventService.subscribeObserver(
            ".Printer.Error", listener, null, 10 * 60 * 1000 );
        leaseRenewalManager = new LeaseRenewalManager(
            observerLease, Lease.FOREVER, null );
    }

    // handle exception subscribing to events
    catch ( Exception exception ) {
        exception.printStackTrace();
    }
}

// receive notification
public void notify( RemoteEvent event ) {
    String output = "\nEVENT: " + ( String ) event.getSource() + "\n";
    SwingUtilities.invokeLater( new TextAppender( printerEventTextArea, output ) );
}

// method main
public static void main( String args[] ) {
    String domain = "";

Fig. 25.13 Management console user interface (part 6 of 7).
We would like `PrinterClientGUI` to receive notifications when printer events occur. Lines 263–283 subscribe to printer events as an observer listener. The event services invokes method `notify` (lines 286–292) when a printer event occurs. Lines 290–291 display the printer event in the lower panel of the GUI. Lines 318–342 define a `private` inner class, `TextAppender` to append text to the Swing container in a thread-safe manner.
Compile `PrinterClientGUI.java` and all other files in its package directory with the command:

```
javac -classpath c:\;c:\jirosdk\lib\jiro.jar
    com\deitel\advjhtp1\jiro\DynamicService\client\*.java
```

Run `PrinterManagementStarter` to instantiate the `PrinterManagement` dynamic service:

```
java -cp c:\;c:\jirosdk\lib\jiro.jar
  -Djava.security.policy=policy.all
    PrinterManagementStarter
domainName
```

where `domainName` is the domain where we deployed the `PrinterManagement` dynamic service.

Now execute the management console (`PrinterClientGUI`):

```
java -cp c:\;c:\jirosdk\lib\jiro.jar
  -Djava.security.policy=policy.all
    PrinterClientGUI
domainName
```

Figure 25.14 shows the resulting output when the user clicks the **Check Status** button.

---

Fig. 25.14 Checking printer status.
Figure 25.15 shows the output on the **Igniter** tool display window after an out-of-paper event occurs.

Figure 25.16 shows the corresponding output on the **PrinterClientGUI** window after it receives notification of an out-of-paper event.
We designed this case study to report exceptions using method `debugException` from class `Debug`. Jiro includes the tools `viewdbg` and `viewdbgw` that provide access to the debug files. Whenever an exception occurs, the `Igniter` error panel displays the location and filename that contains the debug information. The full path name for debug files usually follows the following format:

```
FMA_number\xnumber.debug
```

where `number` represents a number that identifies the debug message.

To view a debug error message, type

```
viewdbg path\filename.debug
```

or

```
viewdbgw path\filename.debug
```

where `path` and `filename` are the path and name of the file which contains the debug information.

### 25.8 Management Policies

The dynamic service we have developed is capable of many tasks. Unfortunately, one task it cannot perform is handling events autonomously. Management policies define how events that occur in a network should be handled. Management policies in Jiro are defined as dynamic services. Developers can customize such dynamic services to respond to events without system-administrator intervention. By defining management policies, network administrators are free to deal exclusively with important events that cannot be automated.

In our example, we would like to use the following management policy

*Printers should always have paper and toner.*

Each management policy should have a single well-defined task. The preceding management policy has two tasks—the printer must always have paper *and* must always have toner. In this case, we should break up the preceding management policy for our printer example into two smaller management policies:

*Printers should always have paper.*

and

*Printers should always have toner.*

The proper definition of a policy gives network administrators fine-grained control over how to manage a network. Furthermore, distributing the management policy to a set of dynamic services allows network administrators to replace and extend individual policies.

Our printer-management example will use two policies to enhance our network-management solution. Class `OutofPaperPolicyImpl` defines the policy, “Printers should always have paper.” `OutofPaperPolicy` (Fig. 25.17) exposes the interface for our out of paper policy dynamic service.

Class `OutofPaperPolicyImpl` (Fig. 25.18) implements `OutofPaperPolicy`. Lines 65–68 register class `PrinterEventListener` with the event service as a respon-
sible listener. Method subscribeResponsibleBefore takes six arguments—the event topic to which this event listener will respond, the responsible listener in front of which to place our responsible listener (null will cause the listener to be placed at the front of the responsible listener list), the subscriber that is the listener we are attempting to register, the object that the event service passes to the listener with each event (may be null) and the lease length (which represents the length for which this listener is requesting to be active). The method returns a Lease object representing the amount of time the event service granted to the listener. To keep our listener active for a longer period than the event service grants, we must renew the lease using a LeaseRenewalManager. Lines 71–72 renew our listener’s lease with Lease.FOREVER. Constructor LeaseRenewalManager takes three arguments—a reference to the lease that will be renewed, the desired expiration for the lease and the lease listener that will receive notifications if exceptions occur during lease renewal (may be null).

Method stopPolicy exposes the stop operation on the out-of-paper policy. Clients may call method stopPolicy to halt the OutofPaperPolicy dynamic service execution. Line 96 cancels the lease on the LeaseRenewalManager object by passing as an argument the lease object to expire. Method notify receives events from the PrinterEventListener. Line 121 obtains the source object, which contains information pertaining to the entity that triggered the event. Line 134 verifies that the source object is an instance of String. If it is not, we know that the event was not fired by our printer. In this case, line 136 throws a NotHandledException that causes the event service to continue propagating the event to its internal list of listeners. If the source object is a String, we verify that the String is of the format that our printer implementation fires (lines 143–144). Line 153 calls method addPaper in the entry-point object of the PrinterManagement dynamic service. Lines 156–172 log a message describing the action taken when handling the out-of-paper event. Lines 200–216 define method getPrinterManagementProxy, which uses our DynamicServiceFinder helper class to find a previously initialized proxy stub for the PrinterManagement dynamic service. Lines 219–228 define method getLookupEntries, which is required for all entry-point objects of a dynamic service.

```java
// OutofPaperPolicy.java
// This class defines the interface for the dynamic service.
package com.deitel.advjhtp1.jiro.DynamicService.policy;

// Java core package
import java.rmi.*;
import java.util.*;

// Jini core package
import net.jini.core.event.*;

public interface OutofPaperPolicy extends RemoteEventListener {
    public void stopPolicy() throws RemoteException;
}
```

Fig. 25.17 OutofPaperPolicy interface.
// OutofPaperPolicyImpl.java
// Handles events generated by printer by registering
// as a responsible listener
package com.deitel.advjhtp1.jiro.DynamicService.policy;

// Java core packages
import java.io.Serializable;
import java.rmi.*;
import java.util.*;

// Java standard extensions
import javax.swing.*;

// Jini core packages
import net.jini.core.event.*;
import net.jini.core.entry.*;
import net.jini.core.lease.*;

// Jini extension packages
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.entry.*;

// Jiro packages
import javax.fma.services.ServiceFinder;
import javax.fma.services.event.*;
import javax.fma.services.log.*;
import javax.fma.util.*;
import javax.fma.server.*;

// Deitel packages
import com.deitel.advjhtp1.jiro.DynamicService.service.*;
import com.deitel.advjhtp1.jiro.DynamicService.common.*;

public class OutofPaperPolicyImpl
    implements OutofPaperPolicy {
    private Lease listenerLease;
    private LeaseRenewalManager leaseRenewalManager;
    private LogService logService;
    private PrinterEventListener listener;
    private PrinterManagement printerManagementProxy;

    // OutofPaperPolicyImpl constructor
    public OutofPaperPolicyImpl()
    {
        // subscribe as an responsible listener to certain event
        listener = new PrinterEventListener( this );
    }
// start the OutofPaper management policy
try {
    // obtain reference to dynamic service entry point
    printerManagementProxy = getPrinterManagementProxy();

    // obtain reference to log service
    logService = ServiceFinder.getLogService();

    // obtain reference to log service
    EventService eventService = ServiceFinder.getEventService();

    // subscribe as responsible listener
    listenerLease =
        eventService.subscribeResponsibleBefore(
            "Printer.Error.OutofPaper", null, listener,
            "OutofPaperEventListener", null, Lease.FOREVER );

    // renew lease indefinitely
    leaseRenewalManager = new LeaseRenewalManager(
        listenerLease, Lease.FOREVER, null );
}

// handle exception starting policy
catch ( Exception exception ) {
    System.out.println( "OutofPaperPolicyImpl: " +
        "Exception occurred when starting policy." );
    System.out.println( "Please read debug file ... \n" );
    Debug.debugException( "starting LowTonerPolicy", exception );
}

System.out.println( "OutofPaperPolicyImpl: started." );

} // end OutofPaperPolicyImpl constructor

// stop OutofPaperPolicyImpl
public void stopPolicy()
{
    // stopping OutofPaper management policy
    try {
        // expire lease
        leaseRenewalManager.cancel( listenerLease );
        System.out.println( "OutofPaperPolicyImpl: stopping." );
    }

    // handle exception canceling lease
    catch ( Exception exception ) {
        System.out.println( "OutofPaperPolicyImpl: " +
            "Exception occurred when canceling lease." );
    }
System.out.println( "Please read debug file ... \n" );
Debug.debugException(
  "stopping OutofPaper policy", exception );
}
}

// receive notification calls
public void notify( RemoteEvent remoteEvent )
  throws UnknownEventException, RemoteException,
  EventNotHandledException
{
  Object sourceObject = null;

  // event source
  try {
    // get event source
    sourceObject = remoteEvent.getSource();
  }

  // handle exception getting event source
  catch ( Exception exception ) {
    System.out.println(  "OutofPaperPolicyImpl: " +
      "Exception occurred when getting event source." );
    System.out.println( "Please read debug file ... \n" );
    Debug.debugException(
      "getting event source", exception );
  }

  // definitely not from our printer
  if ( !( sourceObject instanceof String ) ) {
    throw new EventNotHandledException();
  }

  // obtain String value
  String source = ( String ) sourceObject;

  // verify origin of event
  if ( source.equals( "com.deitel.advjhtp1.jiro." +
      "DynamicService.printer.ErrorMessage=OutofPaper") ) {
    System.out.println( "OutofPaperPolicy: " +
      "handling OutofPaperEvent..." );
  }

  // take action
  try {

    // replenish paper tray
    printerManagementProxy.addPaper( 50 );

    // generate the log message parameters
    Serializable params[] = new Serializable[ 2 ];
params[0] = source;
params[1] = new Date();

// generate localizable message
LocalizableMessage localizableMessage =
    new LocalizableMessage(
        OutofPaperPolicyImpl.class,
        "Action", params, Locale.US);

// generate log message
LogMessage logMessage = new LogMessage(
    localizableMessage, LogMessage.TRACE
    + ".OutofPaperEvent." + source, null);

// log action message
logService.log( logMessage );

} // end try

// handle exception posting log message
catch ( Exception exception ) {
    System.out.println( "OutofPaperPolicyImpl: " +
        "Exception occurred when posting log message." );
    System.out.println( "Please read debug file ...
" );
    Debug.debugException( "log service", exception );
}

} // end if

// not event from our printer
else {

    System.out.println( "OutfPaperPolicy: " +
        " NOT handling OutofPaperEvent..." );

    // responsible listener requirement
    // when not handling event.
    throw new EventNotHandledException();

} // end method notify

// get dynamic services proxies
public PrinterManagement getPrinterManagementProxy()
{
    Entry[] entries = new Entry[] {
        new ServiceInfo( "PrinterManagementImpl",
            "Deitel Association, Inc.",
            "Deitel Association, Inc",
            "1.0", "Model 0", "0.0.0.1" )
    };

Fig. 25.18 OutofPaperPolicy implementation (part 4 of 5).
OutofPaperPolicyImpl requires a properties file to log messages through the log service. See Section 25.6.3 for information on how to create the properties file. Fig. 25.19 shows the contents of that file.

LowTonerPolicy (Fig. 25.20) defines the interface for our second management policy, “Printers should always have toner.” The implementation details are similar to those of management policy OutofPaperPolicyImpl (Fig. 25.18). The differences between the two management policy implementations occur in the information that is logged by the log service and the actions taken when handling events. Fig. 25.20 shows the interface for the LowTonerPolicy dynamic service. Figure 25.21 shows the implementation for LowTonerPolicyImpl.

```
   Action = Added paper to {0} on {1}.
```

Fig. 25.19 OutofPaperPolicyImpl.properties property file for OutofPaperPolicyImpl.

```
   // Fig. 25.20 : LowTonerPolicy.java
   // This class defines the interface for the dynamic service.
   package com.deitel.advjhtp1.jiro.DynamicService.policy;

   // Java core package
   import java.rmi.*;
   import java.util.*;
```

Fig. 25.20 Low toner policy interface (part 1 of 2).
LowTonerPolicyImpl also requires a properties file to log messages through the log service. See Section 25.6.3 for information on how to create the properties file. Figure 25.22 shows the contents of that file.

```java
import net.jini.core.entry.*;
import net.jini.core.lease.*;
import net.jini.core.event.*;
import net.jini.core.entry.*;
import net.jini.core.lease.*;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.entry.*;
import javax.fma.services.*;
import javax.fma.services.event.*;
import javax.fma.services.log.*;
import javax.fma.util.*;
import javax.fma.server.*;
import com.deitel.advjhtp1.jiro.DynamicService.service.*;
import com.deitel.advjhtp1.jiro.DynamicService.common.*;

public class LowTonerPolicyImpl 
    implements LowTonerPolicy {
    private Lease listenerLease;
}
```
private LeaseRenewalManager leaseRenewalManager;

private LogService logService;
private PrinterEventListener listener;
private PrinterManagement printerManagementProxy;

// LowTonerPolicyImpl constructor
public LowTonerPolicyImpl()
{
    // subscribe as an responsible listener to certain event
    listener = new PrinterEventListener( this );

    // start the LowToner management policy
    try {
        // obtain referrence to dynamic service entry point
        printerManagementProxy = getPrinterManagementProxy();

        // obtain reference to log service
        logService = ServiceFinder.getLogService();

        // obtain reference to log service
        EventService eventService =
            ServiceFinder.getEventService();

        // subscribe as responsible listener
        listenerLease =
            eventService.subscribeResponsibleBefore(
                "Printer.Error.LowToner", null, listener,
                "LowTonerEventListener", null, Lease.FOREVER );

        // renew lease indefinitely
        leaseRenewalManager = new LeaseRenewalManager(
            listenerLease, Lease.FOREVER, null );
    }

    // handle exception starting policy
    catch ( Exception exception ) {
        System.out.println( "LowTonerPolicyImpl: " +
            "Exception occurred when starting policy." );
        System.out.println( "Please read debug file ... \n" );
        Debug.debugException( "starting LowTonerPolicyImpl", exception );
    }

    System.out.println( "LowTonerPolicyImpl: started." );
}

// stop OutofPaperPolicy
public void stopPolicy()
{
// stop the LowToner management policy
try {
    // expire lease
    leaseRenewalManager.cancel( listenerLease );
    System.out.println( "LowTonerPolicyImpl: stopping." );
}

// handle exception canceling lease
catch ( Exception exception ) {
    System.out.println( "LowTonerPolicyImpl: " +
                       "Exception occurred when canceling lease." );
    System.out.println( "Please read debug file ... \n" );
    Debug.debugException( "stopping LowTonerPolicyImpl", exception );
}

// receive notification calls
public void notify( RemoteEvent remoteEvent )
    throws UnknownEventException, RemoteException,
           EventNotHandledException
{
    // event source
    Object sourceObject = null;

    // get event source
    try {
        sourceObject = remoteEvent.getSource();
    }

    // handle exception getting event source
    catch ( Exception exception ) {
        System.out.println( "LowTonerPolicyImpl: " +
                             "Exception occurred when getting event source." );
        System.out.println( "Please read debug file ... \n" );
        Debug.debugException( "getting event source", exception );
    }

    // definitely not from our printer
    if ( !( sourceObject instanceof String ) ) {
        throw new EventNotHandledException();
    }

    // obtain String value
    String source = ( String ) sourceObject;

    // verify origin of event
    if ( source.equals( "com.deitel.advjhtp1.jiro." +
                       "DynamicService.printer.ErrorMessage=LowToner" ) ) {

Fig. 25.21 Low toner policy implementation (part 3 of 5).
```java
System.out.println("LowTonerPolicyImpl: handling LowTonerEvent...");

// take action
try {
    // replenish toner cartridge
    printerManagementProxy.addToner();

    // generate the log message parameters
    Serializable params[] = new Serializable[2];
    params[0] = source;
    params[1] = new Date();

    // generate the log message
    LocalizableMessage localizableMessage =
        new LocalizableMessage(
            LowTonerPolicyImpl.class, "Action",
            params, Locale.US);

    // generate log message
    LogMessage logMessage = new LogMessage(
        localizableMessage, LogMessage.TRACE
        + ".LowTonerEvent." + source, null);

    // log action message
    logService.log(logMessage);
}

// handle exception posting log message
catch (Exception exception) {
    System.out.println("LowTonerPolicyImpl:" +
        "Exception occurred when taking action.");
    System.out.println("Please read debug file...\n");
    Debug.debugException("take action", exception);
}

// not event from our printer
else {
    System.out.println("LowTonerPolicyImpl: 
        NOT handling OutofPaperEvent...");

    // responsible listener requirement
    // when not handling event.
    throw new EventNotHandledException();
}
```

Fig. 25.21  Low toner policy implementation (part 4 of 5).
Before we activate our management policies, we must deploy them. For this example, we will restart Jiro with the Clean before Start option checked. This ensures that the previous PrinterManagement dynamic service is not operating when we redeploy the PrinterManagement dynamic service with our new management policies. We must follow the same steps shown in Section 25.7 for both management policies. Due to the dependencies between PrinterManagement dynamic service and the management policies, we will deploy them together.

To deploy the new management solution, review Section 25.7 and perform the following steps:

1. Compile all .java files in the following directories:

```java
// get dynamic services proxies
public PrinterManagement getPrinterManagementProxy()
{
    Entry[] entries = new Entry[] {
        new ServiceInfo( "PrinterManagementImpl",
            "Deitel Association, Inc.",
            "Deitel Association, Inc",
            "1.0", "Model 0", "0.0.0.1" )
    };

    String domain = System.getProperty( "javax.fma.domain" );
    DynamicServiceFinder finder =
        new DynamicServiceFinder( domain, entries );

    // return proxy
    return ( PrinterManagement ) finder.getService();
}
```

Fig. 25.21 Low toner policy implementation (part 5 of 5).

```
Action = Added toner to {0} on {1}.
```

Fig. 25.22 Property file for LowTonerPolicyImpl.
2. Generate the proxy .java file using the jirocw tool for the following classes:

   PrinterManagementImpl
   OutofPaperPolicyImpl
   LowTonerPolicyImpl

3. Compile the newly generated proxy files:

   com\deitel\advjhtp1\jiro\DynamicService\service\n   PrinterManagementImplProxy.java
   com\deitel\advjhtp1\jiro\DynamicService\policy\n   OutofPaperPolicyImplProxy.java
   com\deitel\advjhtp1\jiro\DynamicService\policy\n   LowTonerPolicyImplProxy.java

4. Use rmic to generate the proxy stub for the:

   PrinterEventListener

5. Create PrinterManagementService.jar with the files listed in Fig. 25.23.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>com\deitel\advjhtp1\jiro\dynamicService\common\</td>
<td>DynamicServiceFinder.class</td>
</tr>
<tr>
<td>com\deitel\advjhtp1\jiro\dynamicService\printer\</td>
<td>Printer.class</td>
</tr>
<tr>
<td></td>
<td>PrinterErrorEvent.class</td>
</tr>
<tr>
<td>com\deitel\advjhtp1\jiro\dynamicService\service\</td>
<td>PrinterEventListener.class</td>
</tr>
<tr>
<td></td>
<td>PrinterEventListener_Stub.class</td>
</tr>
<tr>
<td></td>
<td>PrinterManagement.class</td>
</tr>
<tr>
<td></td>
<td>PrinterManagementImpl.class</td>
</tr>
<tr>
<td></td>
<td>PrinterManagementImplProxy.class</td>
</tr>
<tr>
<td>com\deitel\advjhtp1\jiro\dynamicService\service\resources\</td>
<td>PrinterManagementImpl.properties</td>
</tr>
</tbody>
</table>

**Fig. 25.23** Contents of PrinterManagementService.jar (part 1 of 2).
6. Create `PrinterManagementService-ifc.jar` with the files listed in Fig. 25.24.

7. Create `PrinterManagementService-dl.jar` with the files listed in Fig. 25.25.

8. Create `PrinterManagementService-impl.jar` with the files listed in Fig. 25.26.

9. Use the `jarpackw` tool with the command-line arguments in Fig. 25.27.

```
Directory          Files
com\deitel\advjhtp1\jiro\DynamicService\policy\  
    LowTonerPolicy.class
    LowTonerPolicyImpl.class
    LowTonerPolicyImplProxy.class
    OutofPaperPolicy.class
    OutofPaperPolicyImpl.class
    OutofPaperPolicyImplProxy.class
com\deitel\advjhtp1\jiro\DynamicService\policy\resources\  
    LowTonerPolicyImpl.properties
    OutofPaperPolicyImpl.properties
```

Fig. 25.23 Contents of `PrinterManagementService.jar` (part 2 of 2).

```
Directory          Class Files
com\deitel\advjhtp1\jiro\DynamicService\service\  
    PrinterManagement.class
    PrinterManagementImplProxy.class
```

Fig. 25.24 Contents of `PrinterManagementService-ifc.jar`

```
Directory          Class Files
com\deitel\advjhtp1\jiro\DynamicService\service\  
    PrinterManagement.class
    PrinterManagementImplProxy.class
```

Fig. 25.25 Contents of `PrinterManagementService-ifc.jar`.
Fig. 25.26 Contents of PrinterManagementService-impl.jar.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-pool</td>
<td>%JIRO_CLASSPATH%</td>
</tr>
<tr>
<td></td>
<td>c:\PrinterManagementService.jar</td>
</tr>
<tr>
<td>-ifc</td>
<td>PrinterManagementService-ifc.jar</td>
</tr>
<tr>
<td>-impl</td>
<td>PrinterManagementService-impl.jar</td>
</tr>
</tbody>
</table>

Fig. 25.27 Command line arguments for jarpackw (part 1 of 2).
10. Deploy the dynamic services in the Jiro station using the `jardeploy` tool with the command-line arguments in Fig. 25.28.

We have now deployed our two management policies with the `PrinterManagement` dynamic service. Now we must start our `PrinterManagement` dynamic service. Run `PrinterManagementStarter` (Fig. 25.11) to instantiate the `PrinterManagement` dynamic service remotely. Next, we will start the deployed management policies. Class `PoliciesStarter` (Fig. 25.29) instantiates `OutofPaperPolicyImpl` and `LowTonerPolicyImpl` remotely.

Lines 28–30 define the `StationAddress` where our management policies should be instantiated. Lines 34–37 instantiate both management policies—`OutofPaperPolicy` and `LowTonerPolicy`.

---

### Fig. 25.27 Command line arguments for `jarpackw` (part 2 of 2).

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-dl</code></td>
<td><code>PrinterManagementService-dl.jar</code></td>
</tr>
</tbody>
</table>

---

### Fig. 25.28 Command line arguments for `jardeploy`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-station</code></td>
<td><code>SharedJiroStation</code></td>
</tr>
<tr>
<td><code>-domain</code></td>
<td><code>domainName</code></td>
</tr>
<tr>
<td><code>-impl</code></td>
<td><code>PrinterManagementService-Impl.jar</code></td>
</tr>
<tr>
<td><code>-dl</code></td>
<td><code>PrinterManagementService-dl.jar</code></td>
</tr>
<tr>
<td><code>-verbose</code></td>
<td></td>
</tr>
<tr>
<td><code>-inventory</code></td>
<td></td>
</tr>
</tbody>
</table>

---

### Fig. 25.29 Management policies instantiating utility (part 1 of 3).

```java
// Fig. 25.29 : PoliciesStarter.java
// This application demonstrates how to obtain the proxies of management policies.
package com.deitel.advjhtp1.jiro.DynamicService.client;

// Java core package
import java.rmi.*;
```
import javax.fma.common.*;
import com.deitel.advjhtp1.jiro.DynamicService.policy.*;

public class PoliciesStarter {

    public PoliciesStarter( String domain ) {
        OutofPaperPolicy paperPolicyProxy;
        LowTonerPolicy tonerPolicyProxy;

        // set security manager
        if ( System.getSecurityManager() == null )
            System.setSecurityManager( new RMISecurityManager() );

        // obtain station address
        StationAddress stationAddress =
            new StationAddress( domain, null, null, null, null, null, null, null );

        // get the proxies of management policies
        try {
            paperPolicyProxy =
                new OutofPaperPolicyImplProxy( stationAddress );
            tonerPolicyProxy =
                new LowTonerPolicyImplProxy( stationAddress );
        } catch ( RemoteException exception ) {
            exception.printStackTrace();
        }
    }

    public static void main( String args[] ) {
        String domain = "";
        // get the domain name
        if ( args.length != 1 ) {
            System.out.println(
                "Usage: PoliciesStarter Domain" );
            System.exit( 1 );
        } else
            domain = args[ 0 ];
    }
}

Fig. 25.29 Management policies instantiating utility (part 2 of 3).
To instantiate our PrinterManagement, OutofPaperPolicy and LowTonerPolicy dynamic services, we must:

1. Compile PrinterClientGUI, PrinterManagementStarter and PoliciesStarter in the following directory:

   com\deitel\advjhtp1\jiro\DynamicService\client\*

2. Run class com.deitel.advjhtp1.jiro.DynamicService.client.PrinterManagementStarter using a policy.all policy file. You must include the domainName as an argument to PrinterManagementStarter.


4. Run class com.deitel.advjhtp1.jiro.DynamicService.client.PrinterClientGUI using a policy.all policy file. You must include the domainName as an argument to PrinterClientGUI.

Figure 25.30 shows the output on the Igniter display window when an out-of-paper event occurs.
Figure 25.31 shows the resulting action by the **OutofPaperPolicy** dynamic service when it receives an out-of-paper event notification.

Figure 25.32 shows the information displayed by the **viewlog** tool after running the management policies.

---

**Fig. 25.31** OutofPaperPolicy handling out-of-paper event.

**Fig. 25.32** Log contents after events handled by management policies.
Figure 25.33 shows the result of double-clicking a message log entry in the viewlog tool GUI.

25.9 Closing Notes on the Printer Management Solution

The system developed in this chapter has a complex workflow that can be difficult to follow. Obtaining a solid grasp on the workflow of this system is necessary to understand the elegance of a Jiro solution for network management.

Our system contains several important components. The PrinterManagement component represents our entry point object. PrinterManagementImpl implements the core functionality of PrinterManagement. PrinterManagementImpl can delegate calls to a Printer and return events from a Printer. PrinterManagementImpl is responsible for logging all events it receives from the printer.

Our managed entity is the Printer simulator, which exposes an interface that PrinterManagement represents. Printer simulates three events and posts them to the event service with a corresponding topic—".Printer.Error.OutofPaper" or ".Printer.Error.LowToner" or ".Printer.Error.PaperJam". The event service sends a posted event to all observer listeners (PrinterClientGUI and PrinterManagementImpl) and the first responsible listener in the event service’s list of responsible listeners (OutofPaperPolicyImpl’s listener or LowTonerPolicyImpl’s listener, depending on the event topic).

![Log Message](image)

Fig. 25.33 Detailed log information for a specified entry.
Classes `OutOfPaperPolicyImpl` and `LowTonerPolicyImpl` are the management policy implementations that we deployed to help automate trivial and not-so-trivial tasks. `OutOfPaperPolicyImpl` registers an instance of `PrinterEventListener` (a listener we developed for all printer-related events) as a responsible listener to handle all events with topic `".Printer.Error.OutofPaper"`. The event service sends all `OutOfPaperEvents` from the printer to this instance of `PrinterEventListener`, which delegates the event to `OutOfPaperPolicyImpl`. Class `OutOfPaperPolicyImpl` handles an `OutOfPaperEvent` by invoking method `addPaper` on `PrinterManagementImpl`, then logs a message through the log service reporting which action it took. `LowTonerPolicy` handles all `LowTonerEvents`. `LowTonerPolicyImpl`’s response to events is similar to that of `OutOfPaperPolicyImpl`.

We also created a management console. All events are sent to the management console `PrinterClientGUI`, because it registered for events of topic `".Printer.Error"`. `PrinterClientGUI` displays all information about each event it receives. It also enables users to view the printer status at any time and allows users to cancel pending print jobs, which may be useful when paper-jam events occur. To cancel print jobs, `PrinterClientGUI` remotely invokes method `cancelPendingPrintJobs` of class `PrinterManagementImpl` through the shared station.

The resulting system is an automated, self-contained management system that can log events and respond to those event in an appropriate manner. Figure 25.34 illustrates the general structure of the application.

The power of the Jiro management solution in this case study is its ability to delegate responsibilities elegantly and efficiently. Jiro is not limited to bridging the gap between the system administrator’s console and the device; it extends to providing highly extensible, dynamic management facilities to any network.

### 25.10 Internet and World Wide Web Resources

- **jiro.com/documentcenter/**
  This site contains the Jiro-related documents from the Jiro-technology home page.

- **www.jiro.com/overview/**
  This site provides an overview of Jiro technology.

- **sunjweb-001.jiro.com/faqs/**
  This site contains frequently asked questions about Jiro technology.

- **www.jiro.com/education/recipes/**
  This site contains links to technical analysis of the event service.

- **www.jiro.com/education/recipes/service/**
  This site describes how to declare a class as a dynamic service.

- **www.sce.carleton.ca/netmanage/NMfor90s/SimpleNM.html**
  This article discusses network management for the 1990s.

- **sunjweb-001.jiro.com/cgi-bin/Ultimate.cgi?action=intro**
  This site contains a Jiro discussion forum.
SUMMARY

- Jiro provides infrastructure to develop management solutions for distributed resources on a heterogeneous network.
- Jiro technology can simplify and automate management of distributed resources.

**Fig. 25.34** Printer management solution work flow diagram.

**SUMMARY**

- Jiro provides infrastructure to develop management solutions for distributed resources on a heterogeneous network.
- Jiro technology can simplify and automate management of distributed resources.
• Automated management can monitor and control management to minimize human response when reaction is needed.
• Jiro facilitates the development of management solutions that are easy to deploy and expand.
• Jiro uses a three-tier architecture for management solutions.
• Jiro supports other industry standards, such as SNMP and CIM.
• A Jiro management domain contains the managed network resources and the management services that manage the resources.
• Before starting Jiro, a Jiro management domain name must be specified.
• Staring Jiro also starts the class server, `rmid`, transaction service, shared Jiro station, controller service, log service, event service and scheduling service.
• Jiro provides a lookup service for each management domain, so that the static management services and the dynamic management services can be registered with the lookup service.
• Base services or static services include the controller service, the event service, the log service, the scheduling service and the transaction service. Base services are available to the entire management domain.
• Class `javax.fma.services.ServiceFinder` provides a convenient way to locate static services.
• An event publisher posts events to the event service. An event subscriber receives events from the event service.
• A client or service can either post a log message to the log service or retrieve log messages from the log service, obeying search criteria.
• If an application uses the log service, then a resource file that describes the log-message key and value (message template) is required.
• The scheduling service provides methods to schedule a task to perform on a specific date, schedule tasks to perform on event intervals or schedule tasks to perform repeatedly, based on a calendar.
• Method `newRepeatedDateSchedule` creates a schedule for tasks that will be performed repeatedly, based on a calendar date.
• There are three choices for the later-performance policy—`SchedulingService.NONE`, `SchedulingService.ONE` and `SchedulingService.ALL`.
• Static services are always present in a management domain, but dynamic services are not.
• Jiro provides a management station to support dynamic services distributed throughout the management domain and communicate with managed resources.
• A dynamic service that provides access to a managed resource is called Management Facade.
• To make a dynamic service available in a management domain, the dynamic service provider is required to implement the dynamic service and deploy the dynamic service to a station within a management domain.
• Implementing method `getLookupEntries` declares a class as a dynamic service.
• A class that defines method `getLookupEntries` is called a point object. Such a class is the entry point for the dynamic service.
• Jiro provides tools `jiroc` and `jirocw` to generate the proxy of a service.
• Before a dynamic service can be instantiated, it must be deployed to a station. Jiro provides the tool `jardeploy` to deploy a dynamic service.
Jiro provides the GUI tool **jarpack** and the command-line tool **jarpackw** to create the JAR files used by the **jardeploy** tool.

To use the **jarpackw** tool, three seed JAR files are required—an interface JAR file, an implementation JAR file and a download JAR file.

The interface JAR file contains all the interfaces that a dynamic service implements, as well as all classes the interfaces depend on.

The implementation JAR file contains all the classes that required to run the dynamic service.

The download JAR file contains all the classes that should be available to clients of the dynamic service for downloading purposes.

**TERMINOLOGY**

- **automated management**
- **base services**
- **cancel method of Ticket**
- **centralized management**
- **ClientController class**
- **Common Information Model**
- **Context class**
- **Controller class**
- **distributed resources**
- **download JAR**
- **Federated Management Architecture**
- **getControllerService method of ServiceFinder**
- **getEventService method of ServiceFinder**
- **getLogService method of ServiceFinder**
- **getLookupEntries method**
- **getSchedulingService method of ServiceFinder**
- **getTransactionService method of ServiceFinder**
- **heterogeneous network**
- **igniter**
- **IllegalArgumentException**
- **implementation JAR**
- **interface JAR**
- **interoperable**
- **jardeploy tool**
- **jarpack tool**
- **jarpackw tool**
- **Jiro Technology Software Development Kit**
- **jiroc tool**
- **jirocw tool**
- **LocalizableMessage**
- **log method of class LogSearchCriteria**
- **LogMessage.AUDIT**
- **LogMessage.DEBUG**
- **LogMessage.ERROR**
- **LogMessage.INFO**
- **LogMessage.TRACE**
- **LogMessage.WARNING**
- **management facade**
- **newClientController method of ControllerService**
- **newDateSchedule method of SchedulingService**
- **newDurationSchedule method of SchedulingService**
- **newRepeatedSchedule method of SchedulingService**
- **point object**
- **post method of EventService**
- **search of class LogSearchCriteria**
- **ServiceFinder class**
- **Simple Network Management Protocol (SNMP)**
- **static management services**
- **station**
- **StationAddress class**
- **subscribeObserver method of EventService**
- **three-tier architecture**
- **Throwable**
- **TransactionManager**
- **viewlog tool**

**SELF-REVIEW EXERCISES**

25.1 Fill in the blanks in each of the following statements:

a) Jiro is an implementation of the ________.

b) Five static services provided by Jiro are ________, ________, ________, ________, and ________.
c) Class ________ provides a convenient way to locate the static services.
d) A ________ file must be defined before posting a log message to the LogService.
e) A dynamic service must have method ________ declared in its class.
f) To use the jarpackw tool, three seed JAR files are required: ________ , ________, and ________ .
g) Jiro provides a tool ________ to deploy dynamic services.

25.2 State whether each of the following is true or false. If false, explain why.
a) Jiro allows cross-platform information management.
b) Management applications wishing to listen to events from the event service must register a RemoteEventListenerImpl and be running a Web server.
c) Devices using the Simple Network Management Protocol can be managed by Jiro.
d) Jiro can manage devices on its own.
e) The Jiro Event Service sends matching events to all responsible listeners registered for those events.

ANSWERS TO SELF-REVIEW EXERCISES

25.1 a) Federated Management Architecture. b) controller service, event service, log service, scheduling service, transaction service. c) ServiceFinder. d) resource. e) getLookupEntries. f) Interface JAR file, implementation JAR file, download JAR file. g) jardeploy.

25.2 a) True. b) False. If a management application registers a RemoteEventListenerImpl with the event service, the need for a Web server is avoided because both the management application and the Jiro station share RemoteEventListenerImpls. If a management application registers its custom RemoteEventListener implementation, then the management application system must run a Web server that will allow the event service to dynamically load the custom RemoteEventListener implementation. c) True. d) False. Developers need to write applications using Jiro to manage devices and services. e) False. The event service sends matching events to the first responsible listener in the responsible listener chain. If that responsible listener throws an EventNotHandledException, then the event service will send the event to the next responsible listener in the chain. The event will propagate down the chain of responsible listeners until a responsible listener does not throw an EventNotHandledException.

EXERCISES

25.3 Describe the similarities and differences between static services and dynamic services.

25.4 Describe the difference between responsible listeners and observer listeners. Why is it important to have two different types of listeners?

25.5 Compare and contrast the PrinterManagement dynamic service and the management policies OutofPaperPolicy and LowTonerPolicy. How are these dynamic services similar? How are they different?

25.6 Modify our Printer simulator to include a method handlePaperJam, which will fix any paper jams in our printer. Modify the PrinterManagement dynamic service to expose this new operation on the printer. Finally, modify PrinterClientGUI to include a button which, when pressed, will call to the new operation through the dynamic service.

25.7 Create and deploy the following management policy:

"Printer will never be inoperational because of a paper jam."

25.8 Lines 52–54 of PrinterManagementImpl.java (Fig. 25.5) create a Printer-simulator thread. In a real-world scenario, a dynamic service would establish a remote connection to this
printer. This connection could be performed via any popular management protocol (SNMP, WEBM, RMI, etc.). For this exercise:

a) Define interface **RemotePrinter** which exposes the management operations of class **Printer** (Fig. 25.8). **RemotePrinter** must extend interface **Remote**. Create a stand-alone application, **RemotePrinterImpl**, that starts a **Printer** thread. Class **RemotePrinterImpl** must implement **RemotePrinter** interface. **RemotePrinterImpl** must post an RMI stub reference of itself in the Jini lookup service of the Jiro domain. Class **RemotePrinterImpl** must delegate any remote method calls to its **Printer** instance.

b) Modify **PrinterManagementImpl** (Fig. 25.5) to retrieve a **RemotePrinter** stub from the Jini lookup service of the local Jiro domain. **PrinterManagement** dynamic service must delegate all printer management operations to stub **RemotePrinter**.

**BIBLIOGRAPHY**


Objectives

- To introduce CORBA (Common Object Request Broker Architecture).
- To introduce the Interface Definition Language (IDL).
- To use CORBA to develop push-model and pull-model applications.
- To understand distributed exceptions.
- To implement the Deitel Messenger application using CORBA.
- To compare CORBA to other technologies for building distributed systems.

*In the master there is a servant, in the servant a master.*
Marcus Tullius Cicero

*Excellence is to do a common thing in an uncommon way.*
Booker T. Washington

*The virtue in most request is conformity.*
Ralph Waldo Emerson

*It is often wonderful how putting down on paper a clear statement of a case helps one to see, not perhaps the way out, but the way in.*
A.C. Benson
## 26.1 Introduction*

To appreciate what Java has brought to the software-development industry in the past few years, one must first appreciate the happenings just beneath the radar screens of most developers prior to Java’s introduction—the gradual acceptance that legacy systems were not going away and that there existed a need to take advantage of these systems. In addition, the explosive growth of the Web has made it critical to develop systems that can be distributed (networked) “out of the box” rather than building the same communications infrastructure repeatedly. Java offered several features that made these systems possible but did not offer higher-level classes, tools or abstractions for building large-grained distributed systems in generic and predictable fashions. Java lacked the ability to make invocations on

*This chapter was co-authored by Carlos Valcarcel of EinTech, Inc.
local objects that, in reality, were invocations on remote objects. This capability would have allowed Java to make standard method calls on objects that appeared local, but were in a different process space. The supporting infrastructure was not available to Java, so vendors in the distributed-object arena began porting their infrastructure products to support Java. The infrastructure that was the best fit with Java turned out to be CORBA.

CORBA stands for the **Common Object Request Broker Architecture**. More accurately, CORBA stands for “glue.” It allows programs written in various languages, with varying implementations running in disparate locations, to communicate with each other as easily as if they were in the same process address space. Conceptually, in the same way objects cooperate to accomplish specific tasks, CORBA describes architecture made of cooperating services.

**Software Engineering Observation 26.1**

*CORBA does not enforce good systems-integration practice; it merely enables the integration of disparate systems.*

CORBA relies on object technology to accomplish this. Encapsulation, inheritance, polymorphism and dynamic binding hide implementation details, making CORBA’s work appear transparent. Transparency is a crucial goal of CORBA. This transparency enables developers and system integrators to define standard services that legacy systems provide and make these services available to other systems that want to use them. Within this transparent infrastructure, a myriad of services are available to other users that need the services. Three areas leveraged by clients using CORBA are invocation transparency, implementation transparency and location transparency.

**Invocation transparency** defines the viewpoint of a client sending a message to a server. The language used to implement a client specifies how a called object receives a message—how arguments are passed on the stack, how the proper method is called, and how return values are delivered. Once a server (an object that is responsible for delivering a collection of related services) is defined within a CORBA framework, a client can make method calls to this CORBA-enabled object similarly as a call to any other object implemented within the target language. This kind of behavior is more naturally understood by developers using an object-oriented language (e.g., Java, C++, and Smalltalk), but is equally valid for developers using a non-OO language (e.g., C, COBOL, or Lisp). CORBA supports a number of object-oriented and non-object-oriented languages (10 altogether)—C, C++, Java, COBOL, Ada, Lisp, Smalltalk, PL/1, Python and IDLscript.

**Implementation transparency** is a encapsulation applied to distributed systems. A client knows three things about a method invocation—the method name, the method parameters (if any) and the method return type. The client is not concerned with how the underlying code processes parameters and ensures returning valid values—even with performance considerations, the client can notify the server as to any performance constraints prior to making any method invocations. A client invokes a method in the target (client’s) language, and the code invoked to satisfy the method call is executed in any language supported by an OMG mapping.

**Location transparency** allows a client to invoke CORBA-enabled code that might execute elsewhere on the network. This builds on implementation transparency—the clients appear to make local calls, but in reality, they call code that executes in different languages outside of the process space where the client is located (e.g., a Java servlet running in New
York communicates with a COBOL legacy database running in Japan. All of this is possible because of various object-technology concepts (encapsulation, inheritance, and polymorphism) used in implementation-independent ways.

The **Object Management Group (OMG)** is the consortium responsible for CORBA. Founded in April 1989, the OMG’s goals are far reaching and have the backing of approximately 800 members. Their main goal is to create a "component-based software marketplace by hastening the introduction of standardized object software."\(^1\) A major step toward the creation of this object marketplace was the defining of the OMG’s Object Management Architecture (OMA).

The **Object Management Architecture (OMA)** distinguishes CORBA from other distributed system technologies. Objects in an object-oriented system cooperate to achieve solutions to domain problems. The OMA (as a reference architecture) defines a system consisting of cooperating services solving a domain problem. The granularity of the solution is larger, but the concept of objects (or components) working together toward a common goal remains the same.

So, how do the goals of systems integration, transparency, and ease of use come together, using a reference architecture in a fashion that allows multilanguage, multiplatform, and multilocation issues to be handled in a consistent and straightforward fashion? The answer is in these three acronyms—IDL, ORB and IIOP.

For objects to appear to speak the same language, they must use a common mapping to their particular language. For code in languages such as COBOL, Java, C, and Python to communicate to one another, an intermediate language must bind them. This intermediate language should be simple to use, easy to learn and have minimal overhead.

**OMG IDL™,** the **OMG Interface Definition Language** (commonly referred to as IDL), allows developers to describe the interface (or API) of the data type they wish to use remotely in a language-independent fashion. IDL is a pure description language—no implementation details are included in IDL files (unless someone deliberately inserts comments describing implementation assumptions). Using a C++-like syntax, developers describe only two things—the interface to an object and any data structures a method call to the object may take or return as values. An IDL compiler takes an IDL file (conventionally ending with `.idl`) and generates the code needed for a client to call a CORBA-compliant object and for the CORBA-compliant object to receive and return values. When we speak of a CORBA-compliant “object” we make no assumptions about whether it is an object in the way Java describes a object or procedural COBOL code that implements the operations (methods) defined in an IDL file in a non-object oriented fashion. The code that implements the operations (i.e., handles client requests to a CORBA object) constitutes a servant. In a non-object-oriented language there is an individual servant for each operation. In an object-oriented language, one servant (object) may implement all of the operations. The client does not know about the implementation issues involved, so a servant can exist for the duration of a method call, for until the server within which the servant is running is shut down or for some duration in between (this, however, is controlled by an object adapter which mediates all access to servants. We discuss this in more detail in Section 26.4).

An IDL compiler creates a number of language-dependent files for each IDL file compiled. A project would need as many IDL compilers as there are target languages in the system to be developed. To develop a Java front-end to a legacy COBOL system, two IDL
compilers are needed—one to generate the client-side Java code, and one to generate the server-side COBOL code. For example, a Java front end would need the following generated client-side Java files to interact with a back-end system named `StockTicker`:

```java
StockTicker.java
StockTickerOperations.java
StockTickerHolder.java
StockTickerHelper.java
_stockTickerStub.java
```

`StockTicker.java` contains the base definition of the remote server. The `StockTickerOperations` source file contains method definitions of the operations that the CORBA object supports. The `_StockTickerStub` class is responsible for client-side communication, while the `StockTickerHelper` file defines a convenience class usable by the client for CORBA-related tasks.

These files hide the invocation mechanism from a caller. `StockTicker.java` and `StockTickerOperations.java` define interfaces for use by implementation classes; `_StockTickerStub.java` defines actual CORBA invocation code to make the remote method call to a server. Files `StockTickerHelper.java` and `StockTickerHolder.java` are additional support files that we discuss later.

The vendor who supplies the server component ships an IDL file that describes the API to their server object. Using the vendor-supplied IDL file and an IDL compiler, a developer would generate the needed client-side files to talk directly to this server-side object.

If a GUI written in C for the `StockTicker` system were to attempt to send messages to server-side objects written in Java (CORBA-enabled Java objects) the following Java files would have been generated by an IDL compiler to allow the Java server object to receive messages from any callers:

```java
StockTicker.java
StockTickerOperations.java
_stockTickerImplBase.java
```

The files `StockTicker.java` and `StockTickerOperations.java` are the same as above (interfaces for use by implementation classes), and `_StockTickerImplBase.java` represents the CORBA code used by the server to accept method calls and deliver values across the network.

In addition to declaring Java interfaces for developer use, both client-side and server-side files defined above implement base classes as proxies—stand-ins for other objects. Proxies allow a client to believe it is sending a message to one object when the client is really sending the message to another object. CORBA defines two related proxies—a stub and a skeleton. A stub is the client-side proxy and a skeleton is the server-side proxy. Both proxies hide the use of the ORB from the client and the server.

The Object Request Broker is the “ORB part” of CORBA. Interoperability is the ORB’s central purpose. All CORBA-enabled objects must use an ORB to make or receive method requests, but those same objects rarely see the ORB. In the examples that follow, note the similarity of the code for connecting various pieces. This code similarity is the OMG at work. The base connectivity CORBA API is the same regardless of the CORBA implementation.
Portability Tip 26.1

Unless you use vendor-specific value-added features under CORBA 3.0, the replacement of an existing ORB installation with an ORB vendor’s new product should not require the modification of existing CORBA code.

When a client invokes an operation on a distributed object, the client’s ORB uses an object reference to complete the invocation request. For example, if the client was written in Java, it behaves as if the reference is to a local version of the object being invoked. The object reference used by the client (supporting the public API of the object defined in the IDL file) contains an *opaque network reference* (opaque because neither the client nor server infer implementation details with the information contained in the reference). A client invokes operations on a distributed object through an object reference, whereas an *Interoperable Object Reference (IOR)* refers specifically to an object reference whose structure is well-understood by ORBs using the OMG-supported protocols (such as IIOP, discussed momentarily). The object reference comes from an *object adapter* pointing to the remote object. The object reference contains three important pieces of information—the distributed object’s location (an address, but not a memory address), a reference to the adapter that created the object reference and an *object ID* for the servant. All CORBA-compliant ORBs understand what object references are (and, for true interoperability, they must understand IORs) and how to use them to connect clients to servants. The final piece needed for interoperability is the creation and parsing of the IOR into the “on-the-wire” protocol to send a request across the network. However, knowledge of IORs is not enough to allow one vendor’s ORB to communicate predictably with another vendor’s ORB. To enable robust, predictable communication, the OMG specified the *Internet Inter-ORB Protocol or IIOP*.

All vendors must support IIOP (*Internet Inter-ORB Protocol*) for their ORBs to be CORBA-compliant. Vendors can support their own protocol as well, but, at a minimum, they must also support IIOP. IIOP is the protocol for how ORBs communicate with each other. The average developer does not need to know anything about IIOP; a CORBA site administrator maintains the appropriate ORB protocol at any particular location. ORBs that expect to communicate with each other must speak the same protocol. If the ORBs support different protocols, custom bridges must be developed to map the protocols to enable communications between nonstandard-protocol ORBs. As the standard ORB protocol, IIOP should be the default protocol for any ORB installation, although IIOP is not the only choice.

IIOP is the implementation of another OMG standard—the *General Inter-ORB Protocol*, or GIOP. GIOP defines the messages needed by ORBs to communicate with each other and the support for the underlying transport mechanism of the platform on which the ORB runs. TCP/IP is the preferred transport—there is also specification support for Novell IPX and OSI. Vendors are free to support other transport mechanisms. Java, with its built-in network support, and CORBA, with its “under-the-covers” transport to connect distributed objects transparently, are complementary technologies.

Every programming language that the OMG supports must have an IDL-to-target-language mapping. Shortly after Java’s introduction, a number of ORB vendors made their own mappings to satisfy the demand for distributed Java applications. The OMG adopted a Java mapping as a CORBA standard in July 1998. When the OMG ratified the official IDL-to-Java mapping, vendors upgraded their products to conform. Conformity to the IDL mappings is an important step to maintaining integrity in the CORBA world. The IDL-to-Java mapping defines how the standard IDL keywords and data types translate into Java
constructs. Java 2 was the first official Sun release to support the OMG mappings directly. We discuss IDL-to-Java mappings shortly.

### 26.2 Step-by-Step

The steps necessary to implement a distributed system using Java and CORBA are:

1. Perform analysis and design by modeling the problem domain, modeling the system domain and defining subsystems using the system domain.
2. Define the IDL by specifying the API of the subsystems to be distributed and specifying any data structures that cross system boundaries.
3. Implement the servant, using files generated by the IDL compiler.
4. Implement a client, using the stub files generated by the IDL compiler.
5. Decide on a method to distribute the servant’s object reference (traditionally done using a Naming Service, but this is not required).
6. Start the servant implementation.
7. Run the client.

**Software Engineering Observation 26.2**

Implementing a distributed system is a nontrivial task. Developers must follow standard software development practices when using CORBA as an enabling infrastructure to bind various legacy systems together.

**Good Programming Practice 26.1**

Several processes and methodologies for modeling problem domains exist. Make sure the model you choose maps to the problem being solved (otherwise, you might find yourself redoing your project.

The OMG’s OMA remains the preferred reference architecture for directing architectural concerns. OMA defines architecture in the context of a collection of cooperating services. Consider the subsystems you declare as sharing responsibilities. There are more opportunities for process reuse if your system defines fundamental services that are usable by other development groups.

Once subsystems have been defined, you want to decide which subsystems to configure as distributed services. If all of the subsystem designs assume distributed/sharable functionality, the migration of subsystems to CORBA becomes much easier. In any case, the public API of the service becomes the interface accessible by clients. In typical OO fashion, the interface revealed in the IDL can be a subset of the available functionality offered by a server. It is not until the implementation of the server (we use Java as the target language) that the developer worries about how much additional functionality (administration, security, etc.) the server is given.

If there are data structures that a server must return to a client (or vice versa), the identification and definition of these data structures must be in the IDL file. This is a bit of a throwback to the way languages such as C define data structures, but the CORBA model makes allows non-object-oriented languages to describe non-object-oriented constructs in a “pseudo-object” fashion. Two CORBA constructs support this mechanism—the **struct**
and the `valuetype`. A client and a server pass IDL `structs` back and forth as raw data for allowing local handling of information. A standard optimization is sending this data directly to the client so the server can handle more demanding requests. The `valuetype` construct is the IDL description of an entity consisting of both data and behavior. A `struct` consists of pure data (and a `struct` compiled into Java defines the instance attributes as `public`), whereas a `valuetype` encapsulates the data and adds any logic for making the object as simple or complex as needed. Section 26.8.5 explains a CORBA `valuetype` in more detail.

**Performance Tip 26.1**

Make all methods transactional. Remember, every method call is a call across the network. The granularity of services should be such that a useful level of functionality is achievable in as few methods as possible (preferably one method call). If you need to get a `Customer`'s street address, city, state and zip code, create an `Address` class that encapsulates this information. The caller can make one call to the `Address`, rather than make five individual method calls.

After declaring the server API in an IDL file, the only thing missing is the actual implementation of the server. The IDL compiler parses the completed IDL file and generates a collection of support files needed by the server to handle incoming requests. Once the support files are created, the developer can now write implementation code; until this time, the only thing the developer has done is declare the interface, attributes and data needed by the server to handle requests. As a first cut, a developer could implement a “dummy” server that simply returns test data so another developer could write a system client (or possibly a test client) to exercise the server. A more typical example is to write a server that wraps around a legacy back end but hides that fact from any clients.

Assume that the declared server API is in an IDL file and an IDL compiler has compiled the file. Now the client uses the generated client-side files needed to speak with the server. The client needs only the IDL-generated files to take advantage of the underlying mechanism that allows communication across the network transparently. The client could instantiate a local version of the server (instead of using a CORBA object reference, they could simply perform a `new ServerImpl()` and send messages directly, but the server would no longer be a distributed object—the server would be running in the client’s process space having lost the low-coupling advantage we originally had by using the glue code located in the generated files.

### 26.3 First Example: `SystemClock`

This example provides a basic service (time) and allows a client to query for the current time. The requirements are:

- Retrieve the current system time.
- Display the current system time in a GUI window.

The `SystemClock` server interface declares a single method `currentTimeMillis` (which mimics the call in class `java.lang.System`). The method returns the system time as a `long`. A `Date` object would have been more appropriate for Java. If this were an example of the CORBA Time Service, a `Universal Time Object` would have been even more appropriate. However, a `long` helps keep this pure-Java example simple (a non-trivial time service factors issues such as network latency).
26.3.1 SystemClock.idl

Figure 26.1 declares the IDL for the SystemClock server. Lines 1 and 2 are single-line comments; they follow standard C++ syntax with the double slash (//) denoting a single-line comment. Line 4 is the first real line of “code” (CORBA does not dictate implementation, so the IDL is not technically code). The module keyword maps a given name directly to a Java package. Nested module names concatenated together create a complete package name. The module name in Fig. 26.1 is clock.

The curly braces at lines 4 and 10 denote scope boundaries of the block. Note that, in IDL, a semicolon always identifies the end of a block (including the module declaration). Java, C and C++ are block-scoped languages and use a similar syntax but do not consider the block itself as a line that requires a semicolon to be complete. A missing semicolon is a syntax error in IDL.

Lines 7-9 declare the interface the SystemClock server. Is the server type actually SystemClock? From the client’s perspective, the answer is yes—the data type of the server is SystemClock. From the server’s perspective, the answer is no—that is not the direct type. The server is a derived class (concrete implementation) of interface SystemClock. The indirection to the client is deliberate, allowing an arbitrary server implementation.

The curly braces at lines 7 and 9 mark the boundary of the interface declaration. Again, notice the semicolon at the end of line 9 used to identify the end of the block.

Line 8 is a method/function/message declaration. Everything declared in IDL is public, so there are no special keywords in IDL interfaces to denote public, private or protected declarations (although metatypes component and valuetype have keywords private and public). The method currentTimeMillis, by definition, is public and returns an IDL long long, which maps to a Java long (a single IDL long maps to a Java int).

The Java IDL compiler, idlj, compiles systemclock.idl with the command line

```
idlj -td c:\src -pkgPrefix clock com.deitel.advjhtp1.idl
-fall systemclock.idl
```

We discuss the Java tool idlj and its command line options shortly. The Java IDL compiler generates the following server-side files after compiling systemclock.idl:

```
SystemClock.java
SystemClockOperations.java
 SistemaClockImplBase.java
```

![Fig. 26.1: IDL definition for server SystemClock.](image)
SystemClock.java and SystemClockOperations.java are interfaces. SystemClock.java (Fig. 26.2) shows interface SystemClock.

Lines 11 and 12 declare interface SystemClock and the three interfaces from which SystemClock inherits. Two of the three are CORBA-defined types from which all CORBA-enabled objects must inherit org.omg.CORBA.Object and org.omg.CORBA.portable.IDLEntity. The third—interface SystemClockOperations (Fig. 26.3)—is generated from the IDL and declares the public operations of the server. SystemClock inherits from SystemClockOperations, but is otherwise empty. SystemClock defines a base class that inherits from SystemClockOperations and the CORBA interfaces mentioned previously—part of the structure needed by derived classes to appear as proper CORBA objects. SystemClockOperations, inheriting from nothing, declares the single method currentTimeMillis originally defined in the IDL. _SystemClockImplBase inherits from SystemClock, another CORBA interface (InvokeHandler), and a base CORBA implementation class named org.omg.CORBA.portable.ObjectImpl (Impl is an abbreviation for “implementation”). The server requires these three object types to inherit the structure and behavior needed to be a valid CORBA-enabled distributed object. The interface InvokeHandler declares method invoke, which _SystemClockImplBase implements. The ORB uses method invoke to call the various SystemClock methods in a generic fashion. The _SystemClockImplBase method invoke is generated by the IDL compiler based on the IDL interface declared in the IDL file systemclock.idl.

26.3.2 SystemClockImpl.java

SystemClockImpl (Fig. 26.4) is the SystemClock interface implementation. Conventionally, the concrete class that implements a CORBA distributed object’s public interface is named <IDL interface name>Impl. This class does not have to declare a main method, but it can.

Lines 7–9 are standard import statements. We use classes outside the java.lang namespace, so we include them in the Java file. A discussion of the classes declared in these imports follows shortly.
Method `main` (lines 62–74) starts the `SystemClock` server. Line 65 instantiates the `SystemClockImpl` object. The `SystemClockImpl` constructor (lines 20–23) is for the startup initialization. We instantiate `SystemClockImpl` like any other Java object and, in this example, we do not appear to use the object for anything (in fact, the object is waiting for an incoming client request). Line 68 creates an object we can lock against and call its thread’s `wait` method (line 72). The use of keyword `synchronized` (line 71) means no other object can access `object` while the `synchronized` block is active. Line 72 stores the object in the thread queue until someone calls `object.notify`. Object `timeServer` could call `notify` within its own scope, but that would potentially cause the server to terminate when it completes handling an incoming call.
// initialize SystemClockImpl object by calling method register
public SystemClockImpl( String params[] ) throws Exception
{
    register( "TimeServer", params );
}

// register SystemClockImpl object with Naming Service
private void register( String corbaName, String params[] )
    throws org.omg.CORBA.ORBPackage.InvalidName,
           org.omg.CosNaming.NamingContextPackage.InvalidName,
           CannotProceed, NotFound
{
    // Check name of service. If name is null or blank
    // do not attempt to bind to Naming Service.
    if ( ( corbaName == null ) ||
         ( corbaName.trim().length() == 0 ) )
        throw new IllegalArgumentException(
            "Registration name cannot be null or blank." );

    // create and initialize ORB
    ORB orb = ORB.init( params, null );

    // register this object with ORB
    orb.connect( this );

    // find Naming Service
    org.omg.CORBA.Object corbaObject =
        orb.resolve_initial_references( "NameService" );
    NamingContext naming =
        NamingContextHelper.narrow( corbaObject );

    // create NameComponent array with path information to
    // find this object
    NameComponent namingComponent =
        new NameComponent( corbaName, "" );
    NameComponent path[] = { namingComponent };

    // bind SystemClockImpl object with ORB
    naming.rebind( path, this );
    System.out.println( "Rebind complete" );
}

// main method to execute server
public static void main( String[] args ) throws Exception
{
    // Create the SystemClock CORBA object.
    SystemClock timeServer = new SystemClockImpl( args );

    // Wait for requests from the outside.
    java.lang.Object object = new java.lang.Object();

Fig. 26.4  Implementation of the SystemClock server (part 2 of 3).
The `SystemClockImpl` constructor can throw exceptions of type `Exception`. For this example, we allow the constructor to throw `Exception` out of expedience and not out of correctness. The incoming argument `params` is an array of `String` that may contain properties needed to run the server. The `register` method takes array `params` and the `String “TimeServer”` to initialize the `SystemClock` service.

Lines 26–59 declare method `register`. Aside from the `SystemClock` service receiving a name (contained in variable `corbaName`) and any useful configuration parameters (`params`), this method can throw one of four exceptions:

- `InvalidName` thrown by the ORB
- `InvalidName` thrown by the Naming Service
- `CannotProceed`  
- `NotFound`  

The ORB or the Naming Service may throw one of these exceptions based on server interactions. In this case, the ORB is part of the process space of the `SystemClock` server. The server instantiates the objects needed to accomplish its tasks and does not know what kind of ORB it uses. As the CORBA standard does not specify the ORB’s implementation the ORB can be either a local object taking care of network connectivity or a proxy to a `daemon` process (a background process running in its own address space) taking care of network connectivity. The Naming Service is a separate process the server registers with to allow clients to find them.

Lines 35–36 throw a Java `IllegalArgumentException` if the incoming argument `corbaName` is `null` or blank (checked at line 33–34). The server cannot register itself for use by others without a name—this error is severe enough that we would not want to continue without one.

Line 39 is a `static` method call to the ORB class requesting the creation of a new ORB object based on the incoming arguments. A valid incoming argument is a command-line option telling the ORB what communications port to use (for example, `ORBInitialPort`). There is also a no-argument version of `init` that returns a default ORB instead of instantiating a newly configured ORB based on incoming properties. The `init` method can have its arguments as `null`.

Line 42 passes the implementation object to the ORB. All access to the server is now the responsibility of the ORB. The ORB (depending on its implementation) is responsible for load balancing, security, custom filters, etc. In addition, the ORB is responsible for the underlying protocol’s basic operation—the particulars of every generated `ImplBase` class will vary based on its IDL definition (the methods being invoked and the object types...
involved can be different per server type), but ultimately the functionality is the same. The ORB calls method \_invoke (found in the \_SystemClockImplBase class); \_invoke takes care of marshaling incoming arguments (converting values into a format suitable for network transport) and unmarshaling any return values (converting values back into a usable form). Once the ORB has the implementation object, the object cannot be accessed except through the ORB.

A server can make itself available to clients in a number of ways. Lines 45–48 show the easiest way to register a distributed object for client access. The SystemClock server (implemented in SystemClockImpl) registers itself with some directory service clients can reference when they look for services. A directory service serves the same purpose a file system does—it maintains a listing of resources and their locations in an accessible form. A standard CORBA directory service is the Naming Service, which lists resources for future use by clients. The act of registration with Naming does not mean a server is performing any tasks; the server is merely making itself available for client use. The service should be prepared to handle incoming requests for functionality needed by a calling client at any moment. The startup sequence for a server, or bootstrapping, can be problematic—how does the server find the Naming Service the first time the server starts up? Is there a Naming Service finder?

The ORB solves the problem for us—method resolve_initial_references knows a select list of services reachable directly from the ORB. The string “NameService” is a standard name defined in the CORBA specification (along with a list of other names\(^3\)). The ORB effectively has a mini-naming service with which it can perform the lookup of base services.

Lines 45–46 retrieve the object reference to the Naming Service. However, for resolve_initial_reference to work with any of a number of different services, the method returns the object reference as an object of type org.omg.CORBA.Object. Lines 47–48 use static method narrow of class NamingContextHelper to convert the returned object reference into an object of the proper type (in this case, NamingContext). The narrow method is the CORBA mechanism to cast one reference type into another safely. The narrow method checks if the interface into which we are trying to cast the target object is supported by the target object. Standard casting does not work with CORBA object references, because the object reference is a proxy to remote information. All Helper classes have a static narrow method that allows us to cast from a parent class to a derived class.

Lines 52–54 create a NameComponent object and place it in an array. For a name to be properly registered (or bound) the resource must set up a naming context. The server is registered in the main (or root) naming context, so the only thing needed by the Naming Service is the server’s name, which the server sends at line 57 (the name was passed in as an argument to register) using rebind. Method rebind enters the NameComponent if it is not in the Naming Service or re-enters the NameComponent if it is in the Naming Service. However, using rebind in this way allows only one instance of the SystemClock to be accessed at a time (the NamingContext has been hard-coded to the incoming corbaName and every running instance will use the same name). The last SystemClock registered is the server to which clients bind.

In summary, the IDL describes the server, the compiler generates the needed Java support files and SystemClockImpl implements the server.
### 26.3.3 SystemClockClient.java

SystemClockClient (Fig. 26.5) represents the client that connects to SystemClock. SystemClockClient’s primary functionality is in method run (lines 58–80). This object connects to the SystemClock service, requests the current time and displays a String in a JOptionPane. Every time the user clicks the OK button, the client requests the latest time from the server and displays the new time. When the user clicks Cancel, the client application exits.

Method connectToTimeServer (lines 29–48) throws the same exceptions (InvalidName, NotFound and CannotProceed) as method register of class SystemClockImpl. The client and the server throw the same exceptions, because the logic the client uses to read from the Naming Service is similar to the logic used by the server to write to the Naming Service.

Line 35 calls static method init of class ORB to create an ORB. The String array params allows the dynamic configuration of the ORB at runtime (which was passed from method main, which received the String array from the command line). When the client obtains the ORG, lines 37–38 call method resolve_initial_references of this ORB to get the object reference to the Naming Service. Lines 39–40 cast the object reference from a CORBA Object to a NamingContext object using method NamingContextHelper.narrow.

```java
package com.deitel.advjhtp1.idl.clock;

import java.text.SimpleDateFormat;
import java.util.*;
import javax.swing.JOptionPane;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.*;

public class SystemClockClient implements Runnable {
    private SystemClock timeServer;

    // initialize client
    public SystemClockClient( String params[] ) throws Exception {
        connectToTimeServer( params );
        startTimer();
    }

    // Client application for the SystemClock example.
    // Fig. 26.5: SystemClockClient.java
    // Client application for the SystemClock example.
}
```

Fig. 26.5  Client that connects to SystemClock (part 1 of 3).
// use NameService to connect to time server
private void connectToTimeServer( String params[] )
    throws org.omg.CORBA.ORBPackage.InvalidName,
            org.omg.CosNaming.NamingContextPackage.InvalidName,
                    NotFound, CannotProceed
{
    // connect to SystemClock server
    ORB orb = ORB.init( params, null );
    org.omg.CORBA.Object corbaObject =
        orb.resolve_initial_references( "NameService" );
    NamingContext naming =
        NamingContextHelper.narrow( corbaObject );

    // resolve object reference in naming
    NameComponent nameComponent =
        new NameComponent( "TimeServer", "" );
    NameComponent path[] = { nameComponent };
    corbaObject = naming.resolve( path );
    timeServer = SystemClockHelper.narrow( corbaObject );
}

// start timer thread
private void startTimer()
{
    Thread thread = new Thread( this );
    thread.start();

}

// talk to server on regular basis and display results
public void run()
{
    long time = 0;
    Date date = null;
    DateFormat format =
        DateFormat.getTimeInstance( DateFormat.LONG );
    String timeString = null;
    int response = 0;

    while( true ) {
        time = timeServer.currentTimeMillis();
        date = new Date( time );
        timeString = format.format( date );
        response = JOptionPane.showConfirmDialog( null, timeString,
                        "SystemClock Example", JOptionPane.OK_CANCEL_OPTION );

        if ( response == JOptionPane.CANCEL_OPTION )
            break; // Get us out of here
    }

    System.exit( 0 );
}
The client must ask the Naming Service for an object reference to the service the client seeks—the `SystemClock` service, named “TimeServer” (declared in the `SystemClockImpl` class).

**Good Programming Practice 26.2**

Store service names in a `.properties` file, rather than hard-code them in a class. Hard-coding implies additional maintenance of the class files when the values change. A `.properties` file allows for configuring an already deployed system, thereby making the fix an administrative issue, not a development issue.

Lines 43–44 creates a `NameComponent` object. In `SystemClockImpl` (Fig. 26.4) we used the `NameComponent` object to register with the `NamingContext` the path to (and IOR of) where the `SystemClock` is located. In this case, `SystemClockClient` uses the `NameComponent` object to ask the `NamingContext` for the location of a service with a particular name. Lines 45–46 store the `NameComponent` object in an array, then pass the array to the `NamingContext` through its method `resolve`. This method returns an object of type CORBA `Object`, so we use `static` method `SystemClockHelper.narrow` to downcast the object reference to the desired derived class. At this point, the client has an active distributed object, and the client calls the distributed object when the user clicks the **OK** button.
26.3.4 Running the Example

Before running the preceding example, make sure that JDK 1.3 is installed on a workstation and the `PATH` environment variable is updated to include the `bin` directory of the JDK 1.3 installation.

The steps to execute the `SystemClock` example are as follows:

1. Compile the IDL file using `idlj`.
2. Implement and compile the server class.
3. Implement and compile the client class.
4. Run a Naming Service.
5. Run the server.
6. Run the client.

To compile the IDL file on the command line, use the `idlj` compiler supplied with the JDK. A partial list of `idlj`’s command-line options are:

- `-f<client | server | all>`
- `-pkgPrefix <module name or IDL type> <prefix to be appended>`
- `-td <output directory>`

The `-f` option to the `idlj` compiler controls the code generation of the stubs and skeletons. Option `-fclient` generates only client-side files, `-fserver` generates only server-side files and `-fall` generates both. The `-pkgPrefix` option generates package names. This option is used with a module name. A module named `modName` compiled with `-pkgPrefix modName prefix` would generate Java files with a package name of `prefix.modName`. The `-td` option directs `idlj` to write the generated files in the specified directory.

For example, if you had your source code in `C:\src`, the command line would look like this (based on the IDL defined in Fig. 26.1):

```
idlj -pkgPrefix clock com.deitel.advjhtp1.idl -td c:\src -fall SystemClock.idl
```

This generates both the server-side and client-side CORBA-Java files. After implementing the client and server, compile the client and server code. The code supplied in Fig. 26.4 and Fig. 26.5 (`SystemClockImpl.java` and `SystemClock-Client.java`) is sufficient for this example.

Java 2 includes `tnameserv`, which is the basic implementation of the CORBA Object Service (COS) Naming Service. The `tnameserv` tool is not intended as a production-ready Naming Service; rather, it serves as a testing tool to ensure that clients and servers communicate properly. The Naming Service must execute before the server (`SystemClockImpl`) can run. Run `tnameserv` as a background process (in Windows, run `tnameserv` in a separate command window; under UNIX, just append an ampersand to the end of the line starting `tnameserv`):

```
tnameserv -ORBInitialPort 1050
```
All the processes started for this example use port 1050 to communicate. The default port for the name server shipped with Java 2 is port 900—however, sometimes that port is available only to administrators.

Start `SystemClockImpl` as a separate process:

```
java com.deitel.advjhtp1.idl.clock.SystemClockImpl
-ORBInitialPort 1050
```

Start `SystemClockClient` in the same way:

```
java com.deitel.advjhtp1.idl.clock.SystemClockClient
-ORBInitialPort 1050
```

When the client displays its GUI, clicking the **OK** button notifies the client to read the time from the server and to display a **String** with the formatted time. The first Java/CORBA example is complete.

### 26.4 Technical/Architectural Overview

The underlying concepts in CORBA discussed so far should be familiar to those developers who are familiar with design patterns. Stubs and skeletons are **proxies** (objects that control access to other objects). Services that return objects are **factories** (objects that create objects, but defer object implementation to derived classes). Factories and proxies can be found everywhere in the CORBA landscape. Those classes and objects that use design patterns hide a multitude of implementation details from developers.

**Good Programming Practice 26.3**

*Design patterns standardize design terms and solutions. Learning design patterns helps developers reuse designs, components, services and frameworks.*

The ORB is the central mechanism of CORBA. All CORBA-enabled objects must have an ORB standing between them and whatever is calling them (Fig. 26.6). One ORB must exist for every object in a CORBA-enabled distributed system.
We can view an Object Request Broker as the back plane (or communication bus) of distributed systems. As a communication bus, the implication is that everything using an ORB can communicate with everything else on the communication bus using an ORB. What advantage is there to using ORBs? ORBs are a flexible construct, but what problems do they solve? The OMG addressed both of these questions in the *Object Management Architecture*.

A typical software system addresses various business needs. *Legacy systems* (i.e., anything system installed earlier than yesterday) are typically separate, distinct and do not share information. They solve a focused problem, even if the solution involves large volumes of data. Over time, as modifications are made to the systems, they become more expensive to enhance. Over the last 30-40 years, these systems became incompatible. The changing hardware, software, and business landscape created a plethora of incompatible systems until the systems-integration field handled this communication problem. The invention of custom solutions to integrate these systems emphasized the need to make the integration less of a duplication of effort and more of an “out-of-the-box” solution. The OMG, together with its member vendors, created the Object Management Architecture (shown in Fig. 26.7).

The Object Management Architecture (OMA) is the OMG’s reference architecture for distributed systems based on the concept of an Object Request Broker. Using object-technology concepts, the OMA defines a plug-and-play framework where publicly defined objects are available for use by any other object or service through an *object broker*. The object broker is a transparent communication mechanism that ensures that objects send messages reliably to one another regardless of their location. The OMA defines an abstraction that hides the fact that various systems use different programming languages or use incompatible versions of the same language.

The Common Object Request Broker Architecture specifies how the Object Request Broker behaves and works using different languages. OMA defines a polymorphic framework of common services that appear similar from the outside (their API), but differ on the inside (their implementation).

ORBs can be implemented in one of two ways—as libraries or as daemons. Neither the client nor the servant are concerned with the means of implementation. The creation of an ORB object hides the underlying implementation. Traditionally, the client uses a library-based ORB, whereas the server uses a daemon-based ORB. This is an implementation detail for system administrators to decide. From the perspective of the running process, nothing changes.

The ORB plays the most fundamental role possible in the OMA. We now examine how both the client and the servant perceive the ORB. The client communicates with the ORB in one of three ways—through a static stub (generated by the IDL compiler), a dynamic interface (using CORBA's dynamic invocation API) or the ORB’s API. Conceptually, an ORB communicates with a servant in three ways—through a static skeleton, a dynamic interface or the servant’s object adapter (which appears as if the ORB directly interacts with servant). When a servant wants to communicate with another servant, the servant “becomes” a client and all its mechanisms become client-based.

The most straightforward way to communicate with an ORB is through the static stubs and skeletons. They contain the necessary communications code and enables static typing based on the defined IDL. Dynamic invocations (either from a client or a servant) consume...
more overhead but are more flexible because they allow developers to control programmatic-ally how remote objects are invoked (Section 27.2 discusses dynamic invocations). Execu-
tion of remote methods through direct ORB calls is possible, but not recommended. The 
indirection that gives CORBA its strength is bypassed and becomes difficult to restore once 
system implementation is under way. The client/servant communicates with the ORB to 
access certain actions that are possible only through the ORB—operations on object refer-
ences and access to the Interface and Implementation Repositories (two data stores of 
object meta-data). Low-level CORBA development of that kind is needed only to imple-
ment infrastructure support, such as drivers and bridges. Figure 26.8 illustrates the ORB 
interactions.56

---

**Fig. 26.7** Object Management Architecture reference model. Courtesy of Object Management Group, Inc.

**Fig. 26.8** ORB request-interface structure. Courtesy of Object Management Group, Inc.
A CORBA concept discussed earlier was that of object adapters—objects that stand between a client and server to control access to the distributed object. The object adapter acts as a “connector” between a client and the servant code that executes when an operation is invoked (the ever-present ORB stands between the client and the object adapter). Prior to CORBA 3.0, the standard object adapter was the Basic Object Adapter, or BOA. The BOA facilitated binding between a client and a server. CORBA 3.0 defined another object adapter called the Portable Object Adapter (POA). The POA replaces the BOA as the object adapter of choice. The BOA design did not meet demands required by Internet applications. When the OMG first specified BOAs, what is now considered standard functionality (i.e., better CORBA portability between vendors) was not perceived as a high-priority issue. In the same way that Java has supplanted various technologies, the CORBA specification deprecates the BOA in favor of the POA.

The Portable Object Adapter serves many purposes, including the ability to separate the availability of the servant from the actual servant. When a client needs a service, the client needs particular functionality at a particular time. Several components interact to make this possible. First, the client has an object reference represented by a CORBA object. Using the object reference, this CORBA object contains information to find the object adapter that created it. The object adapter handles the client’s invocation by deciding which entity (or servant) can handle the invocation and completing the invocation as needed (based on various configuration options set when the object adapter was created). If the client, holding onto the object reference it received when it first connected to the servant, does not actually need to call the servant for some extended period, waiting for the client to use the servant should not penalize the servant. Scalability would suffer, making CORBA an ineffective solution for systems-integration. By separating the servant from the client’s handle to the service, different servant objects (controlled through object lifetime and activation patterns) can handle the method calls made on a service. There are a number of issues wrapped up in that last statement. The indirection gained by using a POA means transparent handling of those issues.

Chapter 27 discusses the dynamic interface that the client and server use. Static stubs (using the Static Invocation Interface or SII) have hard-coded object types to perform their type checking at compile time, whereas dynamic stubs (using the Dynamic Invocation Interface or DII) perform their type checking at runtime.

CORBA services are the baseline services available to all objects sitting on the ORB communication bus. Because the ORB is the center of a CORBA system, CORBA services can assume the existence of an ORB when they are running. There are sixteen services:

1. Naming Service
2. Event Management Service
3. Life Cycle Service
4. Persistent State Service
5. Transaction Service
6. Concurrency Service
7. Relationship Service
8. Externalization Service
9. Query Service
10. Licensing Service
11. Property Service
12. Time Service
13. Security Service
14. Notification Service
15. Trader Service
16. Collections Service

These services are so fundamental that the Enterprise JavaBeans specification dictates the use of four of them as a requirement (naming, security, persistence and transaction). The CORBA services all have standard IDL interfaces that describe the functionality that these services offer. The purpose of standard interfaces fits right in with another OMA goal: pluggable services should have standard mechanisms for access.

The CORBA facilities are a step above the intermediate CORBA services and come in two groups—horizontal and vertical. The horizontal facilities target client-side functionality, the vertical facilities target domain-specific functionality. The horizontal CORBA facilities have only three specifications:

1. Mobile Agents Facility
2. Printing Facility
3. Internationalization Facility

All three are course-grained enough not to be grouped with CORBA services, but abstract enough not to conflict with potential vendor offerings that could be built on top of CORBA services.

The vertical CORBA facilities—also called CORBA Domains—are between the CORBA services and the Applications Objects (Fig. 26.7). They make use of various CORBA services and horizontal facilities and define domain-specific services. Eleven domain task forces define the various business areas:

1. Common Enterprise Models
2. Finance/Insurance
3. Electronic Commerce
4. Manufacturing
5. Healthcare
6. Telecommunications
7. Transportation
8. Life Science Research
9. Utilities
10. C4I (Command, Control, Communications, Computers, and Intelligence)
11. Space
The various OMG committees for different industries are creating, enhancing and voting on dozens of specifications. CORBA Domains are worth the effort of a visit to the OMG site (www.omg.org).

The Applications Objects are the top layer of the OMA. Relatively speaking, developers find the least amount of reusable code here; the objects at this layer address enterprise-specific domain areas. Applications Objects have the functionality not found at the domain layer, facilities layer or the services layer. Custom applications sit securely in the OMA framework, using existing IDL-defined services or brand new services developers have written in response to design requests.

An understanding of the OMA means a head start in developing an architecture, because so many of the pieces needed in most systems are defined in the OMA. Distributed systems are inherently complex and interesting, but the basic concepts are the same regardless of the designed system’s size.

26.5 CORBA Basics

Distributed objects need to be defined so they can be discovered and used by other distributed objects. We define the distributed objects in IDL and use the stubs and skeletons generated by the IDL compiler to mediate the invocations in a consistent manner.

ORB vendors ship IDL compilers with their products. As of Java 1.2, JavaSoft commoditized the OMG libraries for Java by shipping them with the JDK. Having the CORBA libraries available allows Java/CORBA developers to generate their own stubs and skeletons using the Java-supplied IDL compiler idlj. Ideally, the Java stubs generated by the compiler should interoperate with the skeleton code running under another vendor’s ORB (and the stub code from other vendor’s ORBs should interoperate with the Java skeleton code), but test often and be aware of possible incompatibilities.

OMG document formal/99-07-53 defines the IDL-to-Java mapping and covers everything from package names to Helpers to mapping CORBA pseudo objects. Syntactically IDL is similar to C++, but the similarity ends there. Figure 26.9 lists the most frequently used specification mappings.

<table>
<thead>
<tr>
<th>IDL</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>package</td>
</tr>
<tr>
<td>interface</td>
<td>interface</td>
</tr>
<tr>
<td>struct</td>
<td>class</td>
</tr>
<tr>
<td>const</td>
<td>public static final</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>wchar</td>
<td>wchar</td>
</tr>
<tr>
<td>octet</td>
<td>octet</td>
</tr>
<tr>
<td>string</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Fig. 26.9 IDL keywords, types and their mappings to Java keywords (part 1 of 2).
Packages starting with `org.omg` contain the Java packages that comprise the core CORBA infrastructure. CORBA vendors deliver their version of the CORBA libraries with their Java ORB products.

The following example walks through various IDL keywords and their Java counterparts. The IDL in Fig. 26.10 uses many of the IDL keywords so that we can see how the IDL compiler maps the keywords in the generated files listed in Fig. 26.11–Fig. 26.13.

```java
/*
 * Any comments located outside of the module declaration are
 * ignored by the IDL compiler. This multi-line comment does
 * not appear in any of the files generated by idlj.
 */

// This single-line comment is also ignored by the IDL compiler

module maptest {

    // This comment appears in the generated files for StructMap
    struct StructMap {
        // This comment appears at start of the type declarations
        boolean boolValue;
        char charValue;
        wchar wcharValue;
        octet octetValue;
        string stringValue;
        wstring wStringValue;
        short shortValue;
```
File `maptest.idl` starts with two different types of comments—one multi-line block (lines 1–5), and a single-line comment (line 7). These comments do not appear in the generated files created by the IDL compiler. The comments are for the author and maintainers of the `.idl` file as a documentation aid. Comments placed in an IDL file appear in the generated files only if they are within the `module` scope. Therefore, lines 1–5 and line 7 from `maptest.idl` are extraneous to the `.java` files but not to the `.idl` file.

The `module` name `maptest` (line 9) maps directly to a `package` named `maptest` (however, the compiler can handle the prepending of a `package` name using the `-pkg-Prefix` command line option as we did in the `SystemClock` example). The `module` and its curly braces represent the highest level of symbol scope in IDL.

The comment on line 11 appears in the generated file so any documentation appropriate to the Java code can be included (it is within the scope of the `module`). Lines 12–31 declare `struct StructMap`. `StructMap` uses the bulk of the IDL data types to observe how they map to Java data types. Line 31 completes `struct StructMap` with a closing brace and a semicolon.

It would seem, from the table listing the translation points, that IDL does not handle complex types. In fact, `structs` allow for the complex aggregation of primitives and
object references. However, because a `struct` is not really a class (even though it gets converted into one), the compiler generates a `public final` class to prevent the creation of derived classes from the `struct`. A `struct` is a collection of data compiled into a class definition that a remote servant can return to, or receive from, a client at runtime. In both cases, a local copy of the data is received (instead of an object reference).

Most of the primitive IDL-to-Java mappings are straightforward. The IDL compiler maps signed primitives to corresponding signed primitive types in Java; unsigned IDL primitives run the risk of truncation, because they are mapped to their signed Java primitive counterparts (a signed short cannot store as large a value as an unsigned short). The IDL compiler also enforces the clean initialization of the instance variables by setting them to integer zero, 0.0, `false` or `null` (casting where necessary). Notice, in the generated Java code from `StructMap.java` (Fig. 26.11) that the declared instance variables are `public`. The IDL-generated code breaks encapsulation. In fact, everything in IDL is `public`, because C, COBOL and other IDL-mapped languages do not support encapsulation concepts.

Software Engineering Observation 26.3

Write wrapper classes to surround the `structs`-turned-objects as a way of restoring encapsulation. Wrapper classes define objects that mediate access to another object through a well-defined API. Pass the object representing the CORBA `struct` into the wrapper object's constructor when the wrapper object is instantiated.

An `interface` (line 40) declares attributes clients can query and methods that clients can invoke on remote objects. A `struct` represents a real chunk of runtime data, whereas an `interface` represents a remote object that might be in a different process space, but appears as a local object. The `interface` represents the remote object because the IDL compiler does not `implement` the distributed object. The developer must take the generated code and implement the actual servant, using the generated definitions. Two of the generated interface files (`InterfaceNameOperations.java` and `InterfaceName.java`) encourage developers to (1) structure the servant in a way that allows polymorphic access to the servant through the parent interface (`InterfaceName`) and (2) inherit the distributed protocol behavior we need from an abstract class (`_InterfaceNameImplBase`).

The distributed object definition `interface InterfaceName` contains two `attributes`, one constant and four operations. Line 43 declares an `attribute` that can be read and changed while line 44 explicitly defines `attribute roAttribute` to be `readonly`. The `const` keyword on line 45 declares the `constantValue` field unchangeable.

Only an `interface` can have `attribute` declarations. If the `attribute` keyword is used alone, the compiler generates two methods—an `accessor` (get method) and a `mutator` (set method). Using the `readonly` keyword with `attribute` would generate only the accessor. Originally, the accessor methods looked like JavaBean-style `get` methods (`getAge, getAttribute`). In the latest incarnation of the IDL specification, the JavaBeans naming convention is effectively deprecated. Overloaded methods are generated for both the accessor and mutator using the name of the attribute and are different based on their signatures—the accessor method signature for `attribute balance` would return a value while the mutator would take an incoming argument and return nothing. The implication is that CORBA objects cannot be JavaBeans, but that discussion continues in the CORBAComponents section in Chapter 27.
In IDL, the `const` keyword declares a constant. Syntactically, Java declares constant values using the `final` keyword. Idiomatically, the Java keywords `static` and `final` declare constants as there is no point in having multiple copies of an unchangeable value. The IDL compiler generated code complies with the Java idiom (Fig. 26.13, line 14). The IDL compiler also acknowledges the fact that integral values in Java start out as `longs` and so casts the literal to the smaller type.

The mapping of methods, the parameter-passing modes, holders and arrays represent the remaining concepts needed to define server-side objects and the services they provide. Java developers take many of these features for granted as Java defines cross-platform data sizes and mechanisms to invoke operations on objects in the Java Language Specification. CORBA IDL mappings make the same guarantees for data and invocations between any CORBA-supported languages.

Note the use of the `in` keyword in the method declaration in Fig. 26.10, line 48. IDL includes keywords `in`, `out` and `inout` to describe method arguments. A variable declared as `in` has a copy of itself passed to the called method. The scope of a change made by the servant is visible only to the servant (similar to call-by-value). When the method returns, the client will not see any changes. An `out` variable must be a reference to a Java object containing another Java object where the contained Java object may be substituted with another Java object and the change is visible to the client (similar to call-by-reference). A variable declared as `inout` uses both semantics. Java has two value types—primitives and object references. An object reference is, by definition, a call-by-reference. Use of a primitive, on the other hand, is only as a value. CORBA uses `Holder` objects to make the changing of values by a server-side object similar for both primitives and references. A `Holder` can have its values modified and the client can see the modification. In fact, this works similar to Java `Number` objects that wrap primitive values—however, Java `Number` objects are immutable while `Holders` are mutable (modifiable).

Every time a `struct` or `interface` is declared, the IDL compiler generates an associated `Holder` class for use when the `struct` or `interface` is an `inout` or `out` variable. If a servant needs to read a value (whether primitive or object), send the value as an `in` variable. If a servant needs to send a value to a client (whether primitive or object), send the value as an `out` (or `inout`) variable, and use the `Holder` objects to transport them across the network safely. All of the Java primitives have `Holder` classes available for them in the `org.omg.CORBA` package and the IDL compiler generates the proper method signature in the `Operations` interface defining the `Holder` object needed.

Lines 33 and 36 present the two ways to declare arrays in IDL—using the keyword `sequence` and the use of open-and-close square brackets (`[]`). The `sequence` keyword or `[]` are not used alone in either case. The IDL writer must declare a type definition (a `typedef`) using `sequence` or `[]` to declare the array as a simple name. For example, the type `StructMapSeq` is a `sequence` (array) of type `struct StructMap`.

---

**Common Programming Error 26.1**

The symbol `sequence <StructMap>` cannot define an array identifier; the IDL compiler requires a `typedef`-defined identifier for array types or the compiler generates a syntax error. Renaming the array declaration makes the new symbol become an additional object type.

The `sequence` keyword can be used in one of two ways—`bounded` or `unbounded`. When a length is given, the `sequence` is considered bounded (line 34); otherwise, it is
considered unbounded (line 33). In both cases, the IDL-to-Java compiler unrolls the typedef back to an actual array declaration and generates Helper and Holder types for those array types (in this case, StructMapSeqHelper, StructMapSeqHolder, BoundStructMapSeqHelper, and BoundStructMapSeqHolder). We examine the Holders when we discuss parameter passing modes.

The standard square brackets notation also can declare arrays. However, the use of IDL arrays is non-standard to Java. In Java, lengths are not part of the intrinsic definition of a data type. In IDL, the new array (using []) defined using typedef has a bounded length enforced in the stub/skeleton proxies. The typedef keyword defines an array of a particular data type using a new data type name and the array length. A typedef in IDL is similar to a typedef in C and C++—a new data type appears to exist, but the data type is an alias for use by the compiler. Using a typedef allows the IDL developer to define new data types without the implementation issues involved in the declaration. In maptest.idl, (Fig. 26.10) an array of struct StructMap is declared using the keyword typedef, the type to be redefined and the new name, which is surrounded by <> (lines 33–34). The array’s length (its bound) is declared in the name, but is not used anywhere else; the array is bounded.

Figure 26.11 is the listing for StructMap.java—one of the generated files from maptest.idl. Line 1 (and the first line in each upcoming generated file) declares the package name as maptest. Lines 12 and 17 are the comments listed in Fig. 26.10 at lines 11 and 14. IDL authors and maintainers are responsible for including information on the IDL and its intended use, rather than including information on implementation specific details that can change.

The instance variables declared in lines 18–31 of Fig. 26.11 are the declared struct variables from Fig. 26.10. The Java code initializes the instance variables to include the use of explicit casting. The Java class mapped from the struct has two constructors available for object instantiation—a zero-argument constructor, and a constructor with all of the struct’s field types as arguments. Creating objects of this type can have default values (using the zero-argument constructor) or pre-defined values (using the second constructor).

InterfaceNameOperations.java (Fig. 26.12) contains the interface information declared in Fig. 26.10 at lines 40–52. The IDL compiler translated the operations into Java syntax and the comments from the IDL file are visible in the generated code at lines 12, 15, 18 and 22. A Java interface cannot contain any instance variables or implementation code, which makes an interface a perfect choice for declaring the visible structure for the implementation object. Note that InterfaceNameOperations.java does not declare the attributes defined in Fig. 26.10 lines 43–44, except indirectly through the available accessor and mutator (lines 16, 19 and 20). The IDL does not dictate the implementation of the attributes—the IDL dictates that they be available through a particular interface.

```java
1 package maptest;
2
3 /**
4  * maptest/StructMap.java
5 */
```

Fig. 26.11 IDL-generated file StructMap.java (re-formatted for clarity) (part 1 of 3).
public final class StructMap implements org.omg.CORBA.portable.IDLEntity {

  public boolean boolValue = false;
  public char charValue = (char) 0;
  public char wCharValue = (char) 0;
  public byte octetValue = (byte) 0;
  public String stringValue = null;
  public String wStringValue = null;
  public short shortValue = (short) 0;
  public short uShortValue = (short) 0;
  public int longValue = (int) 0;
  public int uLongValue = (int) 0;
  public long longLongValue = (long) 0;
  public long uLongLongValue = (long) 0;
  public float floatValue = (float) 0;
  public double doubleValue = (double) 0;

  public StructMap ()
  {
    // ctor
  }

  public StructMap( boolean _boolValue, char _charValue,
                    char _wCharValue, byte _octetValue, String _stringValue,
                    String wStringValue, short _shortValue,
                    short _uShortValue, int _longValue, int _uLongValue,
                    long _longLongValue, long _uLongLongValue,
                    float _floatValue, double _doubleValue )
  {
    boolValue = _boolValue;
    charValue = _charValue;
    wCharValue = _wCharValue;
    octetValue = _octetValue;
    stringValue = _stringValue;
    wStringValue = _wStringValue;
    shortValue = _shortValue;
    uShortValue = _uShortValue;
    longValue = _longValue;
    uLongValue = _uLongValue;
    longLongValue = _longLongValue;
    uLongLongValue = _uLongLongValue;
  }
}
The final generated file discussed is InterfaceName.java in Fig. 26.13. The declaration of InterfaceName extends three interfaces and declares the constant constantValue at line 14 (otherwise, the interface is empty). Constants always appear in the file named after the interface declared in the IDL file.

When the IDL compiler processes an IDL file, primary responsibilities for the compiler is to generate these two proxy classes (one for the server, and one for the client). These proxies do not contain any application-specific functionality—they handle only connecting the client to the server. The stub, which directly reflects the API of the server declared in the IDL, can have any of the public methods of the server called from it. In the SystemClock example, when the client makes a call to currentTimeMillis, the client calls a method in the stub that handles calling method _invoke in the stub’s abstract parent (org.omg.CORBA.portable.ObjectImpl). ObjectImpl’s _invoke
method marshals any incoming values (in this case there are none), and the stub code takes care of unmarshaling any return values (which in this case is a `long` value). From the client’s perspective, it makes a method call on the server and blocks until the method returns with a value. In reality, the ORB makes the request by calling `_invoke` on the stub to handle the method call. The `_invoke` method calls the proper method on the servant, waits for the method to return and unmarshals any return values.

Prior to CORBA 3, there were two types of static invocations—synchronous and `oneway` (there was also one for dynamic invocation, but more on that later). When a client makes a `synchronous method call` to a server, the client blocks (does not return) until the server’s method completes. The `oneway` modifier declared an invoked method as returning immediately—but did not declare any quality-of-service options. Quality of Service (known as QoS) defines the policy assigned to a particular task allowing the task to complete within a reasonable length of time. The OMG considered QoS so important a separate QoS framework specification exists to allow developers to control QoS at various levels (generally speaking they are ORB level, thread level and object-reference level). Object reference level QoS overrides thread level QoS and thread level QoS overrides ORB level QoS. QoS makes CORBA 3 invocation choices much more symmetrical—they can be synchronous (normal or `oneway`) or asynchronous (callback or polling). A callback is an invocation made from the server, whereas polling is an invocation made from the client—in either case, a method is being invoked based on the needs of the calling object. From an implementation perspective, a callback has more overhead as the callback object (the client) needs to use CORBA libraries to behave as a CORBA server. A polling client has no additional overhead to make repeated calls to a server. The distinction has to do with a met condition (a server alerts a callback object) or an expected condition (a client polls an object to check on state changes).

**Portability Tip 26.2**

The OMG has not yet decided on standard default values for QoS, ORB vendors can define nonportable values. If you use these values, place them in a separate `.properties` file for the system to read on start-up.

```java
package maptest;

/**
 * maptest/InterfaceName.java
 * Generated by the IDL-to-Java compiler (portable), version "3.0"
 * from maptest.idl
 * Monday, May 14, 2001 4:18:09 PM PDT
 */

// the interface declaration for InterfaceName
public interface InterfaceName extends InterfaceNameOperations,
    org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
{
    public static final int constantValue = ( int ) ( 42 );
} // interface InterfaceName
```

Fig. 26.13 IDL-generated file `InterfaceName.java` (re-formatted for clarity).
A synchronous call is a standard method invocation—a client invokes a method and blocks until the call completes. If an IDL method signature includes the `oneway` keyword, the compiler generates code that does not block on the invocation and returns based on the QoS settings. To return immediately, the method must receive only `in` arguments, must not return values and cannot throw exceptions. The caller does not need to wait for a response (it notifies the server something has happened and does not care what happens next) or the client wants to make itself available as soon as possible to other callers. In either case, a synchronous call is client-initiated.

An asynchronous call differs from a synchronous call. In a callback situation, the server can call a client at any time. In many cases, a server calling a client asynchronously works fine (e.g., listeners in the Java event model), but consider when a client needs the server to notify it in a more controlled fashion? The CORBA Asynchronous Method Invocation specification supports both callback and polling models. The callback model supports (with various QoS options) the ability of a servant to call a client arbitrarily at the discretion of the servant. In the polling model, the client decides when to retrieve a possible result based on a call to a `oneway` method. In standard polling, there may be nothing to retrieve, so the client continues polling until it receives a valid value or decides to stop.

As a matter of convention, passing an object reference to another distributed object (especially for the express intent of callbacks) demands declaring its registration method as `oneway`. The `oneway` keyword tells the IDL compiler to generate code that does not block on the invocation waiting for the called operation to return. This nonblocking behavior prevents the registration method from making method calls on the incoming object reference. If neither object is multithreaded, a server calling a client can cause deadlock. Without the use of `oneway`, the client would wait for the server to return, and the server would wait for the client method to return (Fig. 26.14). Section 26.6 presents an explicit example of the use of the `oneway` keyword.

**Good Programming Practice 26.4**

To avoid deadlock, use the `oneway` keyword whenever passing an object reference to a client through a server method.

---

**Fig. 26.14** Deadlock caused by client calling a server that calls the client.
26.6 Example: AlarmClock

The AlarmClock example is a different type of application from the SystemClock example. SystemClock is a typical pull-model application—the client decides when to retrieve information from the server. The AlarmClock example, on the other hand, is a typical push-model application—the server decides when to send information to the client. In AlarmClock, the client sets a server-side “alarm,” then waits for the server to wake it up when the alarm sounds. In this example, the client generates a random sleep time with which to set the alarm clock. When the alarm goes off, the server notifies the client and the awakened client displays a new sleep time and resets the alarm.

26.6.1 AlarmClock.idl

Figure 26.15 is the IDL declaring the two servers.

Line 4 declares the module name, and lines 5 and 9 declare the names of the server types. The module has two interface declarations—the server interface and the callback interface. The main server is AlarmClock, and AlarmListener is the callback definition. Using the keyword const, line 10 defines a constant called NAME for the implementation code when we bind to the Naming Service. AlarmClock method addAlarmListener at line 12–13 adds AlarmListener objects to a list of callback objects—any object holding a reference to an AlarmListener can call method updateTime. The caller sends in an arbitrary name for the server to associate the caller’s object reference with the name as a primary key. An AlarmListener object is a reference to a listener, so we add keyword oneway to make sure the compiler generates nonblocking code for this operation. The object implementing the AlarmListener interface can register itself with AlarmClock and then wait until the alarm expires. Figure 26.16 is the implementation of the “real” server.

```idl
// Fig. 26.15: alarmclock1.idl
// The IDL for the AlarmClock example

module alarm {
    interface AlarmListener {
        void updateTime( in long long newTime );
    };

    interface AlarmClock {
        const string NAME = "AlarmClock";
        oneway void addAlarmListener( in string listenerName,
                                      in AlarmListener listener );
        void setAlarm( in string listenerName,
                        in long long seconds );
    };
}
```

Fig. 26.15 alarmclock1.idl.
26.6.2 AlarmClockImpl.java

In Fig. 26.16, lines 21–52 are the same as method register in SystemClockImpl (Fig. 26.4, lines 26–59). Lines 26–29 check the incoming arguments. Line 32 creates an ORB using the ORB.init factory method, and line 35 connects with the ORB. Lines 38–41 find a reference to a Naming Service. Lines 45–47 create a NameComponent object, and line 50 sends this NameComponent object to the Naming Service using method rebind.

Lines 56–77 declares the addAlarmListener registration method. After the standard argument checks, Line 76 stores the listener the incoming name in a Hashtable, if the name and listener had not been saved. The Hashtable contains the listener name and an AlarmTimer object that sleeps the required number of seconds, then notifies the listener that the alarm has sounded (AlarmTimer inherits from java.util.Timer which handles the threading for calling the client independent of the main thread). Inner class TaskWrapper (registered with the AlarmTimer at lines 90 and 91) calls the listener method updateAlarm in method TaskWrapper.run (lines 143–151) and promptly removes itself as an alarm. Every time a client sets an alarm using method setAlarm (lines 81–92), we create a new TaskWrapper and return to the caller as fast as we can.

```java
// Fig. 26.16: AlarmClockImpl.java
// Implementation of AlarmClock server.

package com.deitel.advjhtp1.idl.alarm;

// Implementation of AlarmClock server.

public class AlarmClockImpl extends _AlarmClockImplBase {

    // list contains name/alarm pairs of
    // registered objects waiting for an alarm
    private Hashtable alarmList = new Hashtable();

    // register AlarmClockImpl object with Naming Service
    public void register( String corbaName, String params[] )
        throws org.omg.CORBA.ORBPackage.InvalidName,
                   org.omg.CosNaming.NamingContextPackage.InvalidName,
                   CannotProceed, NotFound
    {
        if ( ( corbaName == null ) ||
             ( corbaName.trim().length() == 0 ) )
            throw new IllegalArgumentException("Registration name cannot be null or blank");
```
// create and initialize ORB
ORB orb = ORB.init( params, null );

// register this object with ORB
orb.connect( this );

// retrieve reference to Naming Service
org.omg.CORBA.Object corbaObject =
    orb.resolve_initial_references( "NameService" );
NamingContext naming =
    NamingContextHelper.narrow( corbaObject );

// create NameComponent array with path information to
// find this object
NameComponent namingComponent =
    new NameComponent( corbaName, "" );
NameComponent path[] = { namingComponent };

// bind AlarmClockImpl object with ORB
naming.rebind( path, this );
System.out.println( "Rebind complete" );

// method used by clients wanting to register
// as callback/listener objects
public void addAlarmListener( String listenerName,
    AlarmListener listener )
    throws DuplicateNameException
{
    if ( listenerName == null ||
        listenerName.trim().length() == 0 )
        throw new IllegalArgumentException( "Name cannot be null or blank" );
    else

        if ( alarmList.get( listenerName ) != null )
            throw new IllegalArgumentException( "Name is already registered, please choose another" );
        else

            if ( listener == null )
                throw new IllegalArgumentException( "Listener cannot be null" );
            else

                // create new Timer and save it under listener name
                alarmList.put( listenerName, new AlarmTimer( listener ) );

    }

Fig. 26.16 AlarmClockImpl is the AlarmClock server implementation
(part 2 of 4).
// Set an alarm for a client. If client not registered
// throw a runtime exception.
public void setAlarm( String name, long seconds )
{
    // get alarm for particular client
    AlarmTimer timer = ( AlarmTimer ) alarmList.get( name );
    if ( timer == null )
        throw new IllegalArgumentException( "No clock found for the incoming name" );
    else
        timer.schedule( new TaskWrapper(timer.getListener(), seconds), seconds * 1000 );
}

// main method to execute AlarmClock server
public static void main( String args[] ) throws Exception
{
    AlarmClockImpl alarm = new AlarmClockImpl();
    alarm.register( AlarmClock.NAME, args );
    java.lang.Object object = new java.lang.Object();
    // keep server alive
    synchronized( object ) {
        object.wait();
    }
    // Every listener get an AlarmTimer assigned to them.
    private class AlarmTimer extends Timer {
        // The listener this Timer is assigned to.
        private AlarmListener listener;
        public AlarmTimer( AlarmListener l )
        {
            listener = l;
        }
        // Accessor method so we can get to the listener
        // object reference.
        public AlarmListener getListener()
        {
            return listener;
        }
    } // end of private inner class TaskWrapper

Fig. 26.16 AlarmClockImpl is the AlarmClock server implementation (part 3 of 4).
127 // TaskWrapper takes care of calling our clients
128 // when their alarm expires.
129 private class TaskWrapper extends TimerTask {
130
131     // The reference to our listener
132     private AlarmListener listener;
133     private long seconds;
134
135     // TaskWrapper needs to know who to call and
136     // how long was the alarm set (in seconds).
137     public TaskWrapper( AlarmListener l, long s )
138     {
139         listener = l;
140         seconds = s;
141     }
142
143     public void run()
144     {
145         // Go wake them up!
146         listener.updateTime(seconds);
147         // Discard this TaskWrapper. When the client
148         // wants a new alarm we create a new TaskWrapper.
149         this.cancel();
150     }
151 } // end private inner class TaskWrapper
152 } // end class AlarmClockImpl

Fig. 26.16 AlarmClockImpl is the AlarmClock server implementation
(part 4 of 4).

26.6.3 AlarmClockClient.java

Figure 26.17 is a simple GUI for the application. The GUI consists of a JFrame that displays a String informing the user when the alarm has sounded.

1 // Fig. 26.17: ClockClientGUI.java
2 // GUI used by the AlarmClockClient.
3
4 package com.deitel.advjhtp1.idl.alarm;
5
6 // Java core packages
7 import java.awt.*;
8 import java.awt.event.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class ClockClientGUI extends JFrame {
14     private JLabel outputLabel;
15
Fig. 26.17 ClockClientGUI informs the user when the alarm has sounded
(part 1 of 2).
AlarmClockClient (Fig. 26.18) is the client for the AlarmClock service. Lines 22–38 define the AlarmClockClient constructor. Using the system time, AlarmClockClient gives itself a name and, after creating the GUI, passes the name to method connectToAlarmServer (lines 41–68) along with any incoming parameters. AlarmClockClient connects to the AlarmClock the same way SystemClockClient connects to the TimeServer. Line 48 creates an ORB object, using the ORB factory method init. By passing the client reference in as the second argument, AlarmClockClient registers itself with the ORB. The newly instantiated ORB handles mediating all calls made from and sent to the client. The ORB, created as an object separate from the AlarmClockClient, can control access to and from the AlarmClockClient, but only if the ORB has a usable reference to complete invocations started (or ending) in the stub client code. Method init makes the AlarmClockClient ORB-enabled. Using the same ORB, AlarmClockClient can obtain a reference to the Naming Service. As in the example from Fig. 26.5, ORBs have mini-naming services that allow them to bootstrap with a predefined list of external services (naming being one of them). Using the NamingContextHelper.narrow method, line 55 downcasts the CORBA object returned from resolve_initial_references to an object of type NamingContext. Figure 26.18 displays the output when the client and server connected.

NameComponents allow clients to traverse the Naming Service to look for needed services. Lines 58–59 create a NameComponent object that the Naming Service uses to find the AlarmClock service. When the AlarmClockClient calls the Naming Service methods resolve and downcasts the returned object reference into an object of type

```java
// set up GUI
public ClockClientGUI()
{
    super( "Clock GUI" );

    outputLabel =
        new JLabel( "The alarm has not gone off..." );
    getContentPane().add( outputLabel, BorderLayout.NORTH );

    setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    setResizable( false );

    Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();
    setSize( new Dimension( 450, 100 ) );
    setLocation( ( screenSize.width - 450 ) / 2,
                 ( screenSize.height - 100 ) / 2 );
}

// set label's text
public void setText( String message )
{
    outputLabel.setText( message );
}
} // end of class ClockClientGUI
```

Fig. 26.17 ClockClientGUI informs the user when the alarm has sounded (part 2 of 2).
AlarmClock (lines 61–62), the AlarmClockClient can make a remote method call. Line 65 registers the client with the AlarmClock as an AlarmListener, making itself an asynchronous callback object. The AlarmClockClient finishes method connectToAlarmServer after displaying the GUI and setting the alarm through method updateTime (lines 71–80)—this method (declared in the IDL interface AlarmListener) resets the alarm after every call from the server, allowing the client to sleep until the time expires. The main difference between this code and that of SystemClockClient is the client behavior after finding the Naming Service. The AlarmClockHelper.narrow method is the only nonreusable code.

```
// Fig. 26.18: AlarmClockClient.java
// Client of the AlarmClock service

package com.deitel.advjhtp1.idl.alarm;

// OMG CORBA packages
import org.omg.CORBA.ORB;
import org.omg.CosNaming.*;

public class AlarmClockClient extends _AlarmListenerImplBase {

    // Reference to the GUI to be displayed
    private ClockClientGUI gui;

    // Reference to the alarm clock server we connect to
    private AlarmClock alarmClock;

    // Name of this client used by server to make proper callback
    private String name;

    public AlarmClockClient( String params[] ) throws Exception {
        // create displayable name that is unique among VMs
        // running on same computer (or at least randomly unique)
        name = new Long(
            System.currentTimeMillis() % 10000 ).toString();

        // create GUI to display name and
        // number of seconds before alarm expires
        gui = new ClockClientGUI();

        // connect to TimeService
        connectToAlarmServer( name, params );

        // display GUI and wait for user to terminate app.
        gui.show();
    }
}
```

Fig. 26.18 AlarmClockClient is the AlarmClock client (part 1 of 3).
```java
// perform connection to AlarmServer
private void connectToAlarmServer(
    String name, String params[]
) throws org.omg.CORBA.ORBPackage.InvalidName,
    org.omg.CosNaming.NamingContextPackage.InvalidName,
    NotFound, CannotProceed
{
    // connect AlarmClockClient to an ORB
    ORB orb = ORB.init( params, null );

    // connect to Naming Service and find object
    // reference for AlarmClock service
    org.omg.CORBA.Object corbaObject =
        orb.resolve_initial_references( "NameService" );
    NamingContext naming =
        NamingContextHelper.narrow( corbaObject );

    // resolve object reference in naming
    NameComponent nameComponent =
        new NameComponent( AlarmClock.NAME, "" );
    NameComponent path[] = { nameComponent };
    AlarmClock alarmClock =
        AlarmClockHelper.narrow( naming.resolve( path ) );

    // register this object with AlarmClock service
    alarmClock.addAlarmListener( name, this );
    gui.show();
    updateTime( 0 );
}

// The callback method defined in AlarmListener.
public void updateTime( long seconds )
{
    // Make up a length of time to use for the alarm setting.
    int newTime = ( int )( Math.random() * 10.0 ) + 1;
    gui.setText( "Alarm " + name + " came in at " + seconds +
        " seconds. Resetting to " + newTime + " seconds" );
    alarmClock.setAlarm( name, newTime );
}

// main method to execute client
public static void main( String args[] ) throws Exception
{
    // create client
    try {
        AlarmClockClient client = new AlarmClockClient( args );
    }
    // process exceptions that occur while client executes
    catch ( Exception exception ) {
        System.out.println(
            "Exception thrown by AlarmClockClient:" );
    }
```

Fig. 26.18 AlarmClockClient is the AlarmClock client (part 2 of 3).
26.7 Distributed Exceptions

Exceptions in an object-oriented environment are the preferred means of handling error situations. Java’s exception-handling mechanism works at two levels—compile-time and runtime. Developers can handle exceptions at compile-time using `try` blocks or by expanding method signatures to include any exceptions they throw. If a particular exception is so general that code is written solely to provide specialized error recovery, the exception can inherit from `RuntimeException`—the developer then decides explicitly when the exception should be handled. At runtime, the code is already set to handle normal exceptions, and any code that could throw a `RuntimeException` has to be carefully scrutinized prior to being called to minimize the risk of abnormal code termination. We must consider the happenings when Java code throws exceptions across the network. In CORBA, the server with which a client is communicating might be in a different language. Does this result in using antiquated methods to check status codes?

CORBA specifies two exception types—system exceptions and user exceptions. System exceptions are defined for use by CORBA infrastructure and all operations defined using IDL. User exceptions are CORBA exceptions defined using IDL by the developers of a CORBA system. Every call to a distributed object can throw an exception. Every operation on a CORBA object can throw a CORBA `SystemException`—developers then write code that handle these exceptions. However, even runtime exceptions need attention occasionally, so consider writing additional `try` blocks or wrappers around the stub of a distributed object so that the client does not have to know there is additional error handling occurring when an exception is thrown.

Developers can find the exceptions thrown by a CORBA object by looking in the object’s associated IDL file. For example, the Naming Service declares the `rebind` method in `CosNaming.idl`:

```java
void rebind( in Name n, in Object obj )
  raises( NotFound, CannotProceed,
          InvalidName );
```

Fig. 26.18 `AlarmClockClient` is the `AlarmClock` client (part 3 of 3).
The keyword raises maps to the Java keyword throws. Method rebind raises three exceptions—NotFound, CannotProceed and InvalidName. Cos-Naming.idl defines them as:

```idl
enum NotFoundReason {
    missing_node,
    not_context,
    not_object
};

definition exception NotFound {
    NotFoundReason why;
    Name rest_of_name;
};
definition exception CannotProceed {
    NamingContext cxt;
    Name rest_of_name;
};
definition exception InvalidName{
}
```

Note the use of the keyword exception. IDL exceptions map to Java exceptions inheriting indirectly from java.lang.Exception (they inherit directly from User-Exception). The InvalidName exception—a java.lang.Exception subclass—is closer to standard Java practice, but only as a means of specifying a particular problem area. Exception CannotProceed defines additional attributes available to anyone receiving the exception object, but the attributes are all public.

By definition, standard CORBA exceptions map to Java exceptions as final classes. If a CORBA exception inherits from SystemException, it is also a RuntimeException. All CORBA exceptions are available for use by developers in the various org.omg packages shipped with the JDK. Use CORBA exceptions within a server when appropriate; throw CORBA exceptions only when a CORBA problem has surfaced. Define custom distributed exceptions for service-specific problems that need the attention of the remote caller.

Defining custom exceptions in IDL is straightforward. In Fig. 26.19, lines 2–4 declare an exception useful in alerting a caller that a problem occurred when trying to retrieve a customer record. The find operation (line 7) declares its intention to throw a DatabaseException by using keyword raises.

The IDL compiler generates the exception class definition and needs no further work on the developer’s part. The generated DatabaseException class appears in Fig. 26.20.

To make the AlarmClock more client-friendly, we can add user-defined DuplicateNameException to method addAlarmListener (lines 16–18) in Fig. 26.21. Adding the exception to the signature of addAlarmListener also removes keyword oneway, which does not allow the use of out or inout return values or the throwing of exceptions.
interface CustomerHome {
    void find( in long key ) raises ( DatabaseException );
};

public final class DatabaseException
extends org.omg.CORBA.UserException
implements org.omg.CORBA.portable.IDLEntity
{
    public String msg = null;

    public DuplicateNameException ()
    {
        } // ctor
    
    public DuplicateNameException( String _msg )
    {
        msg = _msg;
        } // ctor
} // class DatabaseException

module alarm {
    exception DuplicateNameException {
        string msg;
    }

    interface AlarmListener {
        void updateTime( in long long newTime );
    }

    interface AlarmClock {
        const string NAME = "AlarmClock";

        void addAlarmListener( in string listenerName, in AlarmListener listener )
            raises( DuplicateNameException );

        void setAlarm( in string listenerName, in long long seconds );
    }
} // module alarm

Fig. 26.19  A user-defined CORBA exception (DatabaseException) and an operation capable of throwing the exception (part 2 of 2).

Fig. 26.20  The generated DatabaseException.java file (reformatted for clarity).

Fig. 26.21  alarmclock2.idl is the IDL for the AlarmClock example.
With the addition of the exception to method `addAlarmListener`, clients cannot register to set alarms if they have the same name. This should be a precondition for this method in any case, but the sentiment is the same. The only changes we need to make to method `addAlarmListener` (Fig. 26.16) are shown in Fig. 26.22 (lines 57 and 66). Line 57 adds the `throws DuplicateNameException` clause to match the method declaration in `AlarmClockOperations.java` (the interface generated when we compiled `alarmclock2.idl`). Line 65 performs the duplicate name check, and line 66 throws the exception if there is an existing listener registered using the incoming name.

Adding `DuplicateNameException` to `addAlarmListener`'s signature has an impact on any methods that call it. In this case, method `connectToAlarmServer` (Fig. 26.18) must add `DuplicateNameException` to its method signature. Making this change to the signature of `connectToAlarmServer` requires any callers either to catch the exception or pass it on to their callers (this example catches everything in method `main`).

Extending Java exceptions by making them distributed does not imply that all Java mechanisms have the same additional capabilities. An area not discussed in CORBA is garbage collection—the reclamation of allocated memory. CORBA does not handle the allocation and deallocation of objects directly. CORBA does not dictate implementation, so we discuss garbage collection in the context of Java in a CORBA environment. Non-Java languages have their own mechanisms for memory allocation and deallocation—these mechanisms create objects for remote use or create structs given to another process somewhere “out there.” Java does not force developers to perform memory deallocation. The same features that attracted millions of developers to Java in the first place are still available even when used in conjunction with legacy languages glued together using CORBA.

```java
public void addAlarmListener( String listenerName,
        AlarmListener listener ) throws DuplicateNameException
{
    if ( listenerName == null ||
            listenerName.trim().length() == 0 )
        throw new IllegalArgumentException(
                "Name cannot be null or blank" );
    else

        if ( list.get( listenerName ) != null )
            throw new DuplicateNameException(
                    "Name is already registered, please choose another" );
        else

            if ( listener == null )
                throw new IllegalArgumentException(
                        "Listener cannot be null" );

            // Create a new Timer and save it under the listener name
            alarmList.put( listenerName, new AlarmTimer( listener ) );
}
```

Fig. 26.22 Excerpt from `AlarmClockImpl.java`. 
When structs are instantiated in Java, as we have stated before, all of the instance variables are public, leaving us with an exposed object. When a client sends a struct object as an argument to a remote operation, the skeleton makes a copy on the server side for use as a local object. What happens to the original data object on the client side? As long as some variable has a handle to the reference, the struct object is not deallocated. As soon as no more variables are pointing to the object (the reference count has become zero) the object is garbage collected. Server-side Java behaves the same way—incoming objects (i.e., references or data objects) exist for as long as a server-side resource has a handle to it—otherwise, these objects are marked for garbage collection.

26.8 Case Study: Chat

Chapter 13 presented a straightforward implementation of the RMI Messenger example. This next example ports the RMI Messenger to CORBA using the concepts discussed in the SystemClock and AlarmClock examples. A chat program is a natural application for distributed system technologies like RMI and CORBA. Chat is a basic network application using a central broadcasting point where a collection of clients pushes in messages and the middleware pushes out messages.

The architectural model for CORBA Messenger begins with determining large, transactional, visible (to the user) functionality offered by the application. The Unified Modeling Language calls the list and description of user-visible functionality use cases. CORBA Messenger has the following use cases:

1. Connect: A client finds and connects to a chat server.
2. Disconnect: A client completes a chat session by disconnecting from the chat server.
3. Send message: A connected client creates a chat message and gives the message to the chat server.
4. Receive message: A connected client receives messages delivered by the chat server. All clients connected to the chat server receive the same message.

Testing and Debugging Tip 26.1

Create test objects as soon as possible in the development cycle of the distributed object. Test the load-bearing capabilities expected of the server on a regular basis.

The Naming Service is a fundamental implementation point for a CORBA system. Naming is the standard way clients can find the service(s) they need to accomplish their tasks. The name for the CORBA Messenger chat server is “ChatServer”. When requested, the Naming Service returns the Interoperable Object Reference (IOR) associated with the chat server name (which the server gave the Naming Service when the server was first started).

Another way for a client to find a service is to use a stringified version of the service’s Interoperable Object Reference. As we mentioned earlier, an object reference contains network information needed by an ORB to find a distributed object. When discovered from a Naming Service, the object reference (in the form of an IOR) is valid for the activation lifetime of the remote object through a particular ORB. The stringified IOR is valid as long as
a server is available to handle any invocations (this is subject to the servant’s POA’s quality of service settings). If desired, an external file can save the IOR for future use.

The object reference of every distributed object can be returned as a string for external use by clients using ORB method `object_to_string`, but using the stringified IOR is not a guarantee that the object is available for use. What a stringified IOR does guarantee is a valid object reference for any session where the object is alive (going through the ORB at the location stated in the object reference). However, clients always expect the Naming Service to have the latest path to a particular object, while there can be no guarantees as to the validity of a stringified IOR. This is similar to the process of starting a car—knowing how to hot wire a vehicle means all cars are available for use, but hot wiring is harder than simply using the ignition key of a particular car.

Abstractions such as the IOR give developers more implementation flexibility. At first glance, passing the message as a string to the ChatServer appears sufficient to allow the server to send messages to all clients. However, modeling the problem domain enables us to abstract the string into a Message, allowing us to do more with the value object being passed between client and server (for example, sending the originating chat name along with the message). Making the ChatMessage a struct means all connected clients receive their own copy of the message for local manipulation rather than sharing one central object (lines 6–11 of Fig. 26.23). The simultaneous access of one shared ChatMessage object among all the users of the system would create a severe bottleneck in the system when all of the clients try to read the object’s contents at the same time.

The use of structs in IDL is an ongoing issue from an object-oriented implementation perspective. A struct is useful for sending collections of data across the network due to a design decision such as performance (i.e., sending 10 struct-like objects between a client and a server offers better performance than sending 10 object references as individual data-member accesses). However, when mapping a non-object-oriented construct (e.g., a struct) to an object-oriented target language, the mapping will not be fully object-oriented without raising more issues than are being solved. In the case of structs, the data becomes unencapsulated. CORBA is language-neutral, and IDL is not an implementation language. As mentioned earlier, everything in IDL is public because C, COBOL and other languages do not support encapsulation concepts. If the middleware were a C server, the stubs and skeletons involved would have to know how to map the proper Java object into a proper C struct. This mapping can be handled by the code generated using the IDL compiler. However, making the generated code more complex raises other issues, such as how the object scope can be opened to allow for the marshaling of these now properly encapsulated values. Remember, CORBA is the glue used to enable systems integration, not an object-oriented solution to domain problems.

Developers do not have to do anything with the code generated to represent structs (they just have to be careful with using unencapsulated values). If ChatMessage were going to be a distributed object, we would have to define ChatMessage as an IDL interface, which developers would have to implement.

### 26.8.1 chat.idl

The server needs server-side interfaces to register and unregister a client and deliver messages to all clients. The client uses its own server-side interfaces to accept messages (after registering with the server).
Figure 26.23 defines the two server *interfaces* for the **Chat** application. Lines 29–44 reflect the IDL of the three services (operations) offered in *interface ChatServer*. The nested *module* names *client* (line 13) and *server* (line 24) create new scope levels. Line 33 references type *client::ChatClient* for use with method *registerClient* (the double colons resolve IDL namespace the same way the dot resolves Java package/class namespace). Lines 14–21 define the **ChatClient** and its service of delivering incoming messages to a user. **ChatServer**’s knowledge of the symbol **ChatClient** defines the symbol linkage for the callback. The IDL gives no indication that **ChatClient** needs to know anything but the symbol **ChatMessage**.

```idl
1 // Chat.idl
2 // This file contains the IDL defining the API to the ChatServer
3 // as well as the ChatClient and ChatMessage.
4
5 module corba {
6 struct ChatMessage {
7     // ChatMessage properties
8     string from;
9     string message;
10    }
11 }
12
13 module client {
14    interface ChatClient {
15        // receive new message
16        void deliverMessage( in ChatMessage message );
17        // method called when server shutting down
18        void serverStopping();
19    }
20 }
21
22 module server {
23    interface StoppableChatServer {
24        void stopServer();
25    }
26
27    interface ChatServer {
28        const string NAME = "ChatServer";
29
30        // register new ChatClient with ChatServer
31        oneway void registerClient( in client::ChatClient client );
32
33        // unregister ChatClient with ChatServer
34        void unregisterClient( in client::ChatClient client );
35
36        // post new message to ChatServer
37        void postMessage( in ChatMessage message );
38    }
39 }
40
```

**Fig. 26.23** **ChatServer**, **ChatClient** and **ChatMessage** interface definitions (part 1 of 2).
The IDL compiler generated the following files for server-side use when we compile chat.idl:

- messenger\client\ChatClientImplBase.java
- messenger\server\ChatServerImplBase.java
- messenger\client\ChatClient.java
- messenger\client\ChatClientOperations.java
- messenger\ChatMessage.java
- messenger\server\ChatServer.java
- messenger\server\ChatServerOperations.java

In addition, the compiler generated two class files that are not used directly:

- messenger\ChatMessageHelper.java
- messenger\ChatMessageHolder.java

The ImplBase classes use ChatMessageHelper to read and write from the Stream for sending objects. Developers never see this use of ChatMessageHelper unless they look at the generated files. On the other hand, developers will use ChatMessageHolder directly if they had declared ChatMessage an out variable. (ChatMessage is used only as an in variable, so ChatMessageHolder is unused.)

Note that ChatClient, ChatClientOperations and _ChatClientImplBase seem to refer to a client—from the ChatServer’s perspective, ChatClient is another server (which makes ChatServer a client during calls to ChatClient). We revisit these files in the client-implementation discussion.

The ChatServer is responsible for keeping a list of clients that want to be part of the broadcast of messages filtering through the server. For simplicity, we use a Hashtable with the user name as the key and the object reference to the ChatClient as the value. In general, method registerClient saves the user name and object reference to the ChatClient in the Hashtable, method removeClient removes the object reference using the user name as the key into the Hashtable and method postMessage iterates the list of object references and calls deliverMessage on each registered client.

In Fig. 26.24, lines 11–13 import the standard packages that contain the symbols for the CORBA classes. At line 19, ChatServerImpl inherits from _ChatServerImplBase. Abstract class _ChatServerImplBase implements
interface ChatServer, which extends ChatServerOperations and requires that ChatServerImpl implement the methods defined in interface ChatServerOperations. The implementation of the three methods is straightforward and is not significantly different from the original ChatServerImpl found in Chapter 13 (which used a Vector instead of a Hashtable). Line 25 defines the HashMap to hold the references to registered clients (the HashMap is also instantiated).

Method register (lines 81–106) looks similar to the register methods from other examples. In this case, we are close enough to the top of the call stack that having a RuntimeException thrown keeps us from continuing if we do not successfully bind to the Naming Service.

Software Engineering Observation 26.4
A good habit to get into is placing try blocks at strategic locations for easier recovery to catch thrown RuntimeExceptions (even if that location is main).

At line 41 of method registerClient, we store the reference to the listener, using the client’s stringified IOR as a key in the Hashtable. Method unregisterClient (lines 49–59) removes listeners at line 75. Other than storing (or removing) the IOR of the incoming client, we have kept CORBA use to a minimum.

Method postMessage (lines 62–78) is the only use of the saved IOR. Line 68 retrieves an Enumeration of the clients in the HashSet, and lines 72–75 loops through the clients and calls method deliverMessage on each of them.

```java
1 // ChatServerImpl.java
2 // ChatServerImpl implements the CORBA ChatServer.
3 package com.deitel.messenger.corba.server;
4
5 // Java core packages
6 import java.io.*;
7 import java.util.*;
8 import java.net.MalformedURLException;
9
10 // Java extension packages
11 import org.omg.CosNaming.*;
13 import org.omg.CORBA.*;
14
15 // Deitel packages
16 import com.deitel.messenger.corba.ChatMessage;
17 import com.deitel.messenger.corba.client.ChatClient;
18
19 public class ChatServerImpl extends _ChatServerImplBase {
20
21     // The ORB that connects us to the network
22     private ORB orb;
23
24     // Map of ChatClient references
25     private Map clients = new HashMap();
26
```

Fig. 26.24 ChatServerImpl implementation of the CORBA ChatServer (part 1 of 4).
// construct new ChatServerImpl
public ChatServerImpl( String[] args )
  throws Exception
{
  super();
  register( ChatServer.NAME, args );
}

// register new ChatClient with ChatServer
public void registerClient( ChatClient client )
{
  // add client to Map of registered clients
  String key = orb.object_to_string( client );
  synchronized( clients ) {
    clients.put( key, client );
  }
  System.out.println( "Registered Client: " + key );
}

// unregister client with ChatServer
public void unregisterClient( ChatClient client )
{
  // remove client from Map of registered clients
  String key = orb.object_to_string( client );
  synchronized( clients ) {
    clients.remove( key );
  }
  System.out.println( "Unregistered Client: " + key );
}

// post new message to ChatServer
public void postMessage( ChatMessage message )
{
  Iterator iterator = null;
  // get Iterator for Set of registered clients
  synchronized( clients ) {
    iterator = new HashSet( clients.entrySet() ).iterator();
  }
  // send message to every ChatClient
  while ( iterator.hasNext() ) {
    ChatClient client = ( ChatClient ) ( ( Map.Entry ) iterator.next() ).getValue();
    client.deliverMessage( message );
  }
} // end method postMessage
// Register ChatServerImpl object with Naming Service
public void register( String serverName, String[] parameters )
throws NotFound, CannotProceed,
    org.omg.CosNaming.NamingContextPackage.InvalidName,
    org.omg.CORBA.ORBPackage.InvalidName
{
    if ( serverName == null )
        throw new IllegalArgumentException( 
                    "Registration name can not be null" );

    // Bind ChatServerImpl object to Naming Service.
    // Create and initialize ORB.
    orb = ORB.init( parameters, null );

    // create servant and register it with ORB
    orb.connect( this );

    org.omg.CORBA.Object corbaObject =
        orb.resolve_initial_references( "NameService" );

    NamingContext naming =
        NamingContextHelper.narrow( corbaObject );

    NameComponent namingComponent =
        new NameComponent( serverName, "" );

    NameComponent path[] = { namingComponent };

    naming.rebind( path, this );

    System.out.println( "Server bound to naming" );
}

// notify each client that server is shutting down and
// terminate server application
public void stopServer()
{
    System.out.println( "Terminating server ..." );

    Iterator iterator = null;

    // get Iterator for Set of registered clients
    synchronized( clients ) {
        iterator = new HashSet( clients.entrySet() ).iterator();
    }

    // send serverStopping message to every ChatClient
    while ( iterator.hasNext() ) {
        ChatClient client = ( ChatClient ) iterator.next();

        client.serverStopping();
        System.err.println( "Disconnected: " + client );
    }
}
// create Thread to terminate application after
// stopServer method returns to caller
Thread terminator = new Thread(
    new Runnable() {
        // sleep for 5 seconds, print message and terminate
        public void run() {
            // sleep
            try {
                Thread.sleep( 5000 );
            }

            // ignore InterruptedExceptions
            catch ( InterruptedException exception ) {};

            System.err.println( "Server terminated" );
            System.exit( 0 );
        }
    }
);

terminator.start(); // start termination thread

} // end method stopServer

// main method to execute ChatServerImpl
public static void main( String[] args ) {
    // set up ChatServerImpl object and bind to Naming Service
    try {
        // create ChatServerImpl object
        ChatServerImpl chatServerImpl =
            new ChatServerImpl( args );

        java.lang.Object object = new java.lang.Object();

        // keep server alive
        synchronized( object ) {
            object.wait();
        }

        // process problems setting up ChatServerImpl object
        catch ( Exception exception ) {
            exception.printStackTrace();
            System.exit( 1 );
        }
    } // end method main
}
Method `main` (lines 155–178) creates the server at lines 161–162. Once again, `main` creates an object named `object` (line 164) and calls `object.wait` (line 168) to keep the server alive by keeping `object` sitting in the thread queue. A `try` block surrounds the core logic in `main` to control the handling of any thrown exceptions.

### 26.8.3 DeitelMessenger.java

The following generated files are available for client-side use after the compile of `chat.idl`:

1. `messenger\client\_ChatClientStub.java`
2. `messenger\server\_ChatServerStub.java`
3. `messenger\client\ChatClient.java`
4. `messenger\client\ChatClientOperations.java`
5. `messenger\ChatMessage.java`
6. `messenger\server\ChatServer.java`
7. `messenger\server\ChatServerHelper.java`
8. `messenger\server\ChatServerOperations.java`

However, as we mentioned before, the following generated server-side files are available for the client to accept callback invocations from the server:

1. `messenger\client\_ChatClientImplBase.java`
2. `messenger\client\ChatClient.java`
3. `messenger\client\ChatClientOperations.java`

In addition, the following Java source files were generated and not used explicitly:

1. `messenger\client\ChatClientHelper.java`
2. `messenger\client\ChatClientHolder.java`
3. `messenger\ChatMessageHelper.java`
4. `messenger\ChatMessageHolder.java`
5. `messenger\server\ChatServerHolder.java`

The `<data type name>` `Helper` classes are for the direct manipulation of mechanisms like the ORB’s object stream. The `<data type name>` `Holder` classes are for `out` and `inout` variable use as dictated in the code generated by the IDL compiler.

A standard CORBA client is concerned with connecting to the Naming Service, getting the object reference to a server (i.e., the `ChatServer`) and making calls to the server. In this case, the client also takes on the role of a server. To be a proper CORBA server the client must inherit from `ChatClientImplBase` and implement any methods declared in `ChatClientOperations`. Class `ChatClientImplBase` implements class `ChatClient`, which extends `ChatClientOperations` so the `ChatClient` server implementation has to implement method `deliverMessage`. Figure 26.25, line 23 declares the inheritance to `ChatClientImplBase` which implies the client must implement method `deliverMessage` somewhere. The client is an implementation of
**MessageManager** (line 24)—an interface that represents the connectivity portion of the **DeitelMessenger** GUI client.

```java
// CORBAMessageManager.java
// CORBAMessageManager implements the ChatClient remote
// interface and manages incoming and outgoing chat messages
// using CORBA.
package com.deitel.messenger.corba.client;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// Java extension packages
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

// Deitel packages
import com.deitel.messenger.*;
import com.deitel.messenger.corba.client.ChatClient;
import com.deitel.messenger.corba.ChatMessage;
import com.deitel.messenger.corba.server.*;

public class CORBAMessageManager extends _ChatClientImplBase
    implements MessageManager {

    // incoming ORB configuration parameters
    private String[] configurationParameters;

    // listeners for incoming messages and disconnect notifications
    private MessageListener messageListener;
    private DisconnectListener disconnectListener;

    // ChatServer for sending and receiving messages
    private ChatServer chatServer;

    // CORBAMessageManager constructor
    public CORBAMessageManager( String[] parameters )
    {
        configurationParameters = parameters;
    }

    // connect to ChatServer
    public void connect( MessageListener listener )
    throws Exception
    {
        // find ChatServer remote object
        ORB orb = ORB.init( configurationParameters, null );
    }
}
```

Fig. 26.25 **CORBAMessageManager** implementation of interface **MessageManager** using CORBA (part 1 of 3).
org.omg.CORBA.Object corbaObject = 
  orb.resolve_initial_references( "NameService" );
NamingContext naming = 
  NamingContextHelper.narrow( corbaObject );

// Resolve the object reference in naming
NameComponent nameComponent =
  new NameComponent( ChatServer.NAME, "" );
NameComponent path[] = { nameComponent };
chatServer =
  ChatServerHelper.narrow( naming.resolve( path ) );

// register client with ChatServer to receive messages
chatServer.registerClient( this );

// set listener for incoming messages
messageListener = listener;

} // end method connect

// disconnect from ChatServer
public void disconnect( MessageListener listener )
  throws Exception
{
  if ( chatServer == null )
    return;
  chatServer.unregisterClient( this );
  messageListener = null;
  // notify listener of disconnect
  fireServerDisconnected( "" );
} // end method disconnect

// send ChatMessage to ChatServer
public void sendMessage( String fromUser, String message )
  throws Exception
{
  if ( chatServer == null )
    return;

  // create ChatMessage with message text and user name
  ChatMessage chatMessage =
    new ChatMessage( fromUser, message );
  // post message to ChatServer
  chatServer.postMessage( chatMessage );
} // end method sendMessage

Fig. 26.25 CORBAMessageManager implementation of interface MessageManager using CORBA (part 2 of 3).
The GUI client is not concerned with registering with the Naming Service. **ChatServer** has the client-registration method **registerClient**, so the **ChatClient** (**CorbaMessageManager**) passes itself in and makes itself available to the server.

Let us now look at the chat client as a client. The relevant methods for the client are **connect**, **disconnect** and **sendMessage**.

Method **connect** (lines 43–67) initializes an ORB, gets the Naming Service object reference and asks for the **ChatServer** reference from the Naming Service. Line 62 passes the client to the server for callback invocations (as in the **AlarmClock** example—view the client as a listener, the **ChatServer** as an event generator and the **ChatMessage** as an event object passed to the listener). If we execute the call to method **registerClient** successfully, the client is now ready to receive messages from any clients connected to the **ChatServer**.

Method **deliverMessage** (lines 101–106), which implements the operation declared in **chat.idl**, takes the incoming message, and sends it to objects registered with the **MessageManager** to receive messages. There is no indication of CORBA at work except for the direct access of fields **from** and **message** at lines 104–105. Knowing that **ChatMessage** is an IDL **struct** allows us to understand why we are violating encapsulation, but does not make the transgression any easier to accept. The good news is that
changing either of the two fields has no impact on the other clients receiving the message (the object is local), but leaves open the possibility of another object in any particular chat client changing the information in an uncontrolled fashion. From a design perspective, ChatMessage should be a read-only (immutable) object. Once the server has sent a message, there is no reason to update the message; the other clients cannot see the change. A possible solution is to wrap the ChatServer in a client-side proxy object (which would allow us to handle client-side CORBA exceptions in a transparent way) and have the ChatServer proxy instantiate a read-only wrapper around the ChatMessage before sending it to the chat client. The implementation issues become problematic at that point, however, because the proxy would have to register itself with the real ChatServer as the callback target instead of the client registering itself with the ChatServer. The client would become the callback target of the chat server proxy when the proxy decides to call the client’s deliverMessage method.

DeitelMessenger (Fig. 26.26), as the entry point to the client-state implementation, controls the instantiation of the CORBAMessageManager (lines 15–16) and the ClientGUI objects (line 19).

### 26.8.4 Running Chat

The steps for running this example are as follows:

1. Compile the IDL file using idlj—for example (all on one line):
   ```
   idlj –pkgPrefix chat com.deitel.advjhtp1.idl –td c:\src -fall chat.idl
   ```
2. Implement and compile the server class (Fig. 26.24).
3. Implement and compile the client class (Fig. 26.26).
4. Open a window and run the tnameserv Naming Service:
   ```
   tnameserv –ORBInitialPort 1050
   ```
5. Open a window and run the server:
   ```
   java com.deitel.messenger.corba.server.ChatServerImpl
   –ORBInitialPort 1050
   ```
6. Open a window and run the client (then open another window and run another client):
   ```
   java com.deitel.messenger.corba.client.DeitelMessenger
   –ORBInitialPort 1050
   ```

### 26.8.5 Issues

CORBAMessenger was a straightforward port of RMIMessenger. The differences between the two versions are minimal, with the largest being the use of a CORBA struct compared to the use of a serialized object. The number of remote objects, the operations and the result (broadcasting messages to registered clients) are identical in both cases.

Why use a local object for the messages received by the clients? How would making ChatMessage a distributed object alter the application’s complexity?
The current schema of a `ChatMessage` object is a `struct` holding two strings. For the client to use this object that contains two `public` data members with no accessor/mutator methods, it creates a `String` that accesses first the instance variable `from`, then variable `message`. If this `struct` were changed into an `interface` and the `from` and `message` fields turned into `interface attributes`, the advantages are the following:

1. The proper encapsulation of `ChatMessage`. `ChatMessage` would not have to reveal its complexity to any callers.

2. `ChatMessage` would be a remote object. All clients, once given the `ChatMessage` object reference, could inspect the message object and would see changes to the object immediately. In addition, the `ChatMessage` data would not have to be unmarshaled upon arrival to the client.

3. The client would have a clearly defined API with which to work. Having no `public` data members means the object can properly offer services through its operations rather than direct data access.

4. All connected clients could share a single `ChatMessage`. `ChatMessage` would be a direct open pipeline to the clients. If the server makes a change, the clients see the change immediately and can access the message when appropriate.

The disadvantages of turning `ChatMessage` into an `interface` (i.e., a true distributed object) are the following:

```java
// DeitelMessenger.java
// DeitelMessenger uses ClientGUI and CORBAMessageManager to
// implement an CORBA chat client.
package com.deitel.messenger.corba.client;

// Deitel packages
import com.deitel.messenger.*;

public class DeitelMessenger {

    // launch DeitelMessenger application
    public static void main( String args[] ) throws Exception {
        // create CORBAMessageManager for communicating with server
        MessageManager messageManager =
            new CORBAMessageManager( args );

        // configure and display chat window
        ClientGUI clientGUI = new ClientGUI( messageManager );
        clientGUI.setSize( 300, 400 );
        clientGUI.setResizable( false );
        clientGUI.setVisible( true );
    }
}
```

**Fig. 26.26** `DeitelMessenger` application for launching the CORBA chat client.
1. **ChatMessage** would be a remote object. As a distributed object, all the registered clients would share a single **ChatMessage**, creating a massive network bottleneck at the server. If a particular object has a substantial amount of information to share with clients, consider putting the information inside a **struct**. This is important because of one fact—during the access of a distributed object, there is marshaling of data, protocol mapping, and unmarshaling of data when the call completes (if the operation has data to return). The overhead to deliver a **ChatMessage struct** (a complete round-trip network call) occurs once. The overhead of a distributed **ChatMessage** starts with the first network call to receive the object reference, accessing the **from** attribute involves another network call and accessing the **message** attribute yet another, bringing the total number of network calls up to three. **ChatMessage**, as a **struct**, has trivial network overhead when it arrives—the unmarshaling of the data. Additional overhead is the creation of an object to hold the data, but this represents a fixed startup cost in addition to the single network call. The more data in the object, the greater the advantage to having a local copy.

2. All connected clients could share **ChatMessage**. In many situations, having numerous clients sharing a distributed object is a valid practice. However, in this case, not giving the clients their own copy means dealing with data locking issues. When do we know that no one is accessing the message? Behavior similar to Java **synchronized** is advantageous, but CORBA does not directly support this behavior (the developer is always free to programmatically support this behavior). In addition, giving others the ability to change shared data (or data with the potential to be shared) without some straightforward mechanism for notifying interested parties of the change is not a good idea.

 Sending a local object to a client circumvents all of the preceding problems.

**Performance Tip 26.2**

*Use structs or valuetype whenever passing data around from a client to a server or vice versa. An object reference is convenient for communication purposes, but can have significant overhead if used solely for the transmission of data. Not only are structs and valuetype an optimal way to pass around presentation information, but the distributed service that generated the data can also continue to respond to service requests in a timely fashion.*

In RMI, when an object is serializable, it is streamed across the network and reassembled in the JVM of the caller. The reason to send an object by value is the same as the reason to use structs—a local copy of an object has less access overhead than a remote object that needs repeated access. A serialized object in RMI is a real object with all of the standard advantages inherent to an object. A CORBA **struct** has no behavior. The target language of the mapped **struct** adds some overhead (in a non-object-oriented target language, the overhead is almost nonexistent), but beyond that, the ORB adds nothing. Structurally, all IDL **structs** mapped to Java classes implement `org.omg.CORBA.portable.IDLEntity` but do not inherit behavior or structure from any classes (except for `java.lang.Object`). Is there a CORBA mechanism that works like RMI, but maintains the language neutrality of CORBA? The answer is “yes.”

The **Objects-by-Value (OBV)** specification defines a new kind of **interface** type—a **struct** with behavior semantics (the ability to support the declaration of operations)
called a **valuetype**. The advantage of **valuetype**s is the ability to pass copies of “objects” around a distributed system that contain behavior in addition to data. This maps indirectly with Java serialized objects, yet continues to support language neutrality. As a matter of record, the OBV specification had to support the concept of objects-by-value in all the target languages approved by the OMG.

Implementing objects-by-value is not a trivial undertaking. Declaring a **valuetype** is straightforward, as is seen in Fig. 26.27, lines 7–17.

To create a Java **valuetype** object, define the **valuetype** in an IDL file and compile the definition using **idlj** (or a vendor-supplied IDL compiler). Derive a new class from the base **valuetype** class definition, and make sure there is a no-argument constructor as well as a constructor capable of receiving each of the fields belonging to the **valuetype** defined in the IDL. In addition to the generated **valuetype** files, the compiler creates a `<valuetype>DefaultFactory` source file. If necessary, subclass the `<valuetype>DefaultFactory` generated by the IDL, and add any addition operations included in the IDL file. Use of an object instantiated from a **valuetype** is the same as the use of any other Java object. Figure 26.28 lists some keywords added to IDL to support **valuetype**s.

```
// Fig. 26.27: Chat.idl
// This file contains the IDL defining the
// API to the ChatServer as well as the
// ChatClient and ChatMessage.

module obvcorba {
    valuetype ChatMessage {
        // ChatMessage properties
        private string from;
        private string message;
        string getSenderName();
        string getMessage();
        factory create( in string from, in string message );
    }

    module client {
        interface ChatClient {
            // receive new message
            void deliverMessage( in ChatMessage message );
        }
    }

    module server {
        interface ChatServer {
            const string NAME = "ChatServer";
        }
    }
}
```

Fig. 26.27 **chat.idl** with ChatMessage changed to be a **valuetype** (part 1 of 2).
The keyword `private` is misleading—`private valuetype` fields become `protected` Java instance variables. This allows the derived class to access the variables, but outside callers cannot (at least non-package-level outside callers. Java scope rules allow objects defined in a package to access the `protected` fields of the other objects defined in the same package.)

Generated code handles everything else. The server creates the `valuetype` object the way the server always creates objects (using `new` or factory methods, but using the derived class definition). The client receives a `valuetype` object and treats it as a local object. The stubs and skeletons call the `DefaultFactory` when a `valuetype` arrives. `ChatMessage`, instead of being complete when we compile the IDL, is now a base class for us to derive. `ChatMessageImpl.java` (Fig. 26.29) is the class definition to use to instantiate a `valuetype` object.

```
31 // register new ChatClient with ChatServer
32 void registerClient( in string chatName,
33       in client::ChatClient client );
34
35 // unregister ChatClient with ChatServer
36 void unregisterClient( in string chatName );
37
38 // post new message to ChatServer
39 void postMessage( in ChatMessage message );
40 }
41 }
42 }; // end of module obvmessenger
```

**Fig. 26.27** `chat.idl` with `ChatMessage` changed to be a `valuetype` (part 2 of 2).

**Fig. 26.28** Keywords specific to `valuetypes`.

---

```
1 // Fig. 26.29: ChatMessageImpl.java
2 package com.deitel.messenger.obvcorba;
3
```

**Fig. 26.29** `ChatMessageImpl` is the `ChatMessage` implementation (part 1 of 2).
ChatServerImpl does not change. The server receives an object of type ChatMessage (i.e., ChatMessageImpl) and sends that object to the registered clients. DeitelMessenger has two changes at lines 92–93:

```
ChatMessage message = new ChatMessageImpl( userName, messageToSend );
```

and at lines 245–246:

```
messageArea.append( "\n" + message.getSenderName() + 
   " > " + message.getMessage() );
```

Everything else happens via standard CORBA mechanisms, so DeitelMessenger and ChatServerImpl objects continue to work as written.

At CORBAMessageManager time of this writing, the idlj delivered with JDK 1.3.0_02 does not support the valuetypes as specified in the OBV specification. The code presented works under the Java 2 Software Development Kit. Updated code examples will be available at www.deitel.com when idlj complies more closely with the OBV specification.
26.9 Comments and Comparisons

Comparing the RMI version of the chat example to the CORBA version raises a couple of interesting questions not the least of which is the use of a `Hashtable` instead of a `Vector` in the CORBA version of the server. Using a `Vector` is much simpler than using a `Hashtable` (but bear in mind that this is an implementation detail. The type of object used to hold a collection can change many times during the design of a sub-system). In this particular case, `Hashtable`s allow a lookup by primary key, whereas `Vector`s do not. Primary key lookup is important because of the way RMI communicates with remote objects compared to CORBA. In the beginning, RMI required both stubs and skeletons to communicate, but switched over to a `skeleton-optional` implementation protocol in JDK 1.2. When a client passes its remote object reference, the client-side Java object that contains the RMI object reference communicates directly with the server (in effect, “pretending” to be the server’s skeleton code). The underlying RMI mechanism matches the remote object reference with the Java object on the server-side that contains the object reference.

CORBA, on the other hand, has the `stub` communicating with the `skeleton` before going to the server. The CORBA object reference is always the same, but the Java object holding the reference is different. The skeleton is responsible for unmarshaling all incoming arguments, so a Java object is always created for non-primitive argument values. Also, the current version of JavaIDL supports only BOAs and does not support POAs, which would have offered slightly different behavior. In RMI, the Java object is always identical for each client because there is no skeleton between the stub and the server.

More fundamentally, is this the best design for a chat application? OMA has the final answer—use services whenever possible. CORBA specifies a `Notification Service` that delivers messages synchronously and asynchronously. Using the Notification Service as the messaging middleware makes the chat-application implementation even simpler.

CORBA is a powerful means of designing robust distributed systems. Legacy applications have become valuable long-term assets, and new applications have become reusable services that can exchange data in a platform-neutral fashion. IIOP allows ORBs to communicate in a predictable and robust way, and the Object Management Architecture gives developers a roadmap to follow as they design and implement systems to accomplish tasks of substantial complexity. The OMG continues to push the boundaries of system design and integration with specifications in areas where commonality of purpose and design give developers more leverage to build larger systems. In Chapter 27, we present an in-depth look at system architecture, including CORBA services, the CORBA Component Model, Enterprise JavaBeans and an assessment of CORBA and RMI.

26.10 Internet and World Wide Web Resources

The JavaIDL Home Page lists various documents relating to JavaIDL and Javasoft’s implementation of various OMG specifications.

[http://www.omg.org](http://www.omg.org)
The Object Management Group home page. This is the ultimate portal for CORBA information. The OMG home page lists membership information as well as information on the latest happenings in the distributed systems world.
www.omg.org/technology/documents/formal/
object_management_architecture.htm
Download the Object Management Architecture Specification from this HTML page as either a PDF or a postscript file.

www.omg.org/technology/documents/formal/index.htm
OMG Formal Documentation. All of the approved specifications in the various areas the OMG is involved in are at this location. The latest CORBA, CORBA services and IDL mapping specifications have their own link.

www.omg.org/technology/documents/formal/corbaiiop.htm
The CORBA 2.4 specification describes the mappings and functionality required for a product to be considered CORBA-compliant.

www.omg.org/technology/documents/formal/
.omg_idl_to_java_language_mapping.htm
The OMG IDL-to-Java Mapping Specification explains how IDL-described services translate into Java. This specification is the definitive source for mapping information.

www.omg.org/technology/documents/formal/naming_service.htm
The Naming Service Specification describes the baseline functionality needed by a distributed naming service to be useful to a distributed architecture. The specification explains definitions and Naming Service IDL in detail.

This link has the February 1998 paper Objects-by-Value, Joint Revised Submission - w/Errata.

www.omg.org/cgi-bin/doc?formal/98-12-09
This link on the Object Management Group Web site has the CORBA services: Common Object Services Specification document.

www.corba.net
The CORBA.net site, sponsored by the OMG and the Distributed Systems Technology Centre in Australia, showcases CORBA technology in concrete ways, including a web-based application showing interoperability between different vendor’s ORBs.

Open Source ORBs

openorb.exolab.org
ExoLab Group’s OpenORB is an open source implementation of the CORBA specification. OpenORB is at CORBA 2.4.1 compliance and includes many of the standard tools developers expect from a full-fledged CORBA product.

www.cs.wustl.edu/~schmidt/TAO.html
The Ace ORB (TAO). The ACE project is a collection of frameworks and CORBA concepts implemented in an open source ORB reflecting high quality and best practice.

CORBA Vendors
The following vendors sell ORBs or products built on ORBs. The following is not a comprehensive list.

www.iona.com
Iona.

www.inprise.com/visibroker
Inprise.

www.capeclear.com
Cape Clear.
**www.vertel.com**
Vertel.

**CORBA Services Vendors**

**www.prismtechnologies.com**
Prism Technologies is a commercial CORBA vendor that has implemented many of the defined CORBA services.

**www.ooc.com**
Object Oriented Concepts, Inc. (an Iona company) is also a commercial ORB vendor. OOC’s free (for non-commercial use) download includes a handful of CORBA services with their ORBacus product.

**openorb.exolab.org/services.html**
The same ExoLab Group folks that implemented OpenORB also have implemented many of the standard CORBA services. These CORBA services are available for free download.

**SUMMARY**

- CORBA stands for the Common Object Request Broker Architecture.
- The reason CORBA exists is to allow programs written in various languages, with varying implementations running in disparate locations, to communicate with each other as easily as if they were in the same process address space.
- Transparency is a crucial goal of CORBA.
- Invocation transparency defines the viewpoint of a client sending a message to a server.
- Implementation transparency is a standard object concept (encapsulation) applied to distributed systems.
- Location transparency allows a client to invoke CORBA-enabled code that could be executing anywhere on the network without worrying about where the method call is happening.
- The Object Management Architecture (OMA) is one of the defining features that distinguish CORBA from other distributed system technologies.
- The OMA (as a reference architecture) defines a system consisting of cooperating services solving a domain problem.
- IDL, the Interface Definition Language, allows developers to describe the interface (or API) of the data type they wish to use remotely in a language-independent fashion.
- IDL is a pure description language—no implementation details are included in IDL files.
- An IDL compiler creates a number of language-dependent files. A project would need as many IDL compilers as target languages in the system to be developed.
- Any CORBA-enabled objects must use an ORB to make or receive method requests, but those same objects rarely see the ORB.
- When a client using an ORB connects to a distributed object, the ORB returns an object reference to the object in question.
- CORBA defines an object reference as being an Interoperable Object Reference (IOR). The IOR contains three important pieces of information—the distributed object’s location (an address, but not a memory address), a reference to the adapter that created the IOR and an object ID for the servant.
- IIOP is the standard ORB protocol that all vendors must support for their ORB’s to be considered CORBA-compliant.
IIOP is the concrete implementation of another OMG standard—the General Inter-ORB Protocol, or GIOP. GIOP defines the messages needed by ORBs to communicate with each other as well as support for the underlying transport mechanism of the platform on which the ORB is running.

Java 2 was the first official JavaSoft release to support the OMG mappings directly.

The steps necessary to implement a distributed system using Java and CORBA are: 1) perform analysis and design, 2) define the IDL, 3) implement the servant using files generated by the IDL compiler, 4) implement a client using the stub files generated by the IDL compiler, 5) execute a CORBA Naming Service, 6) execute the servant implementation, 7) execute the client.

IDL follows standard C++ syntax with the double slash (//) denoting a single-line comment.

The `module` keyword maps a given name directly to a Java package. Nested `module` names concatenated together create a complete package name.

Curly braces denote the boundaries of a scope block and always end with a semicolon.

The keyword `interface` defines a CORBA-enabled object.

Everything declared in IDL is public so there are no special keywords to denote public, private or protected declarations.

Conventionally, the concrete class that implements the interface that defines the public API to a CORBA distributed object is named `<interface name>Impl`.

Once the ORB has the implementation object, no one accesses the object except through the ORB.

A standard CORBA directory service is the Naming Service. It exists for the sole purpose of listing resources for future use by clients.

Method `resolve_initial_references` knows a select list of services reachable directly from the ORB. The ORB effectively has a mini-naming service with which it can perform the lookup of base services.

For a name to be properly registered (or bound, in CORBA terminology) the resource must set up a naming context using a `NameComponent` object.

The `tnameserv` tool is a basic implementation of the CORBA Object Service (COS) Naming Service.

We can think of an Object Request Broker as the back plane (or communication bus) of distributed systems.

The Object Management Architecture is the OMG’s reference architecture for distributed systems based on the concept of an Object Request Broker.

The OMA defines a plug-and-play framework where publicly defined objects are available for use by any other object or service through an object broker.

The Common Object Request Broker Architecture defines how the Object Request Broker behaves and how it works using different languages.

ORBs can be implemented in one of two ways—as libraries, or as standalone processes known as daemons.

The client can talk to the ORB either through a static stub, a dynamic interface or directly.

An ORB can talk to a servant in one of three ways—through a static skeleton, a dynamic interface or directly.

Object adapters are objects that stand between a client and server to control access to the actual distributed object.

CORBA 3.0 defined an object adapter called the Portable Object Adapter (POA).
• The Portable Object Adapter serves a number of purposes, including the ability to separate the availability of the servant from the actual servant itself.

• Static stubs (using the Static Invocation Interface or SII) have hard-coded object types to allow for compile-time type checking while dynamic stubs (the Dynamic Invocation Interface or DII) perform their type checking at runtime.

• CORBA services are the baseline services available to all objects sitting on the ORB communication bus.

• The ORB is the central core of a CORBA system, so the CORBA services can assume the existence of an ORB when they are running.

• The CORBA services all have standard IDL interfaces that describe what the various services offer in the way of functionality.

• The CORBA facilities are a step above the intermediate CORBA services and come in two groups—horizontal and vertical.

• The horizontal facilities target client-side functionality.

• The vertical CORBA facilities—also called CORBA Domains—are between the CORBA services and the Applications Objects.

• The Applications Objects are the top layer of the OMA. They have functionality not found at the domain layer, facilities layer or the services layer.

• Distributed objects need to be defined in a manner that allows them to be discovered and used by others distributed objects.

• OMG document formal/99-07-53 defines the IDL-to-Java mapping and covers everything from package names to helpers to mapping CORBA pseudo objects.

• The Java packages that constitute the core CORBA infrastructure are located in the packages that start with org.omg.*.

• Comments placed in an IDL file appear in the generated files only if they are within the module scope. Any other comments are purely for the maintainer of the IDL file.

• A struct is the definition of a collection of data compiled into a class definition that a remote servant can return to or receive from a client at runtime.

• Most of the primitive IDL-to-Java mappings are straightforward. Signed primitives stay that way (except there is no keyword int in IDL); unsigned IDL primitives run the risk of truncation when they are mapped to their signed Java primitive counterparts (a signed short cannot store as large a value as a unsigned short).

• The IDL compiler also initializes the instance variables by setting them to integer zero, 0.0, false or null (casting where necessary).

• There are two ways to declare arrays in IDL—using the keyword sequence or by the use of open-and-close square brackets.

• The sequence keyword can be used in two ways—bounded and unbounded. When a length is given, the sequence is considered bounded, otherwise it is considered unbounded.

• The more standard square brackets notation can also declare arrays. However, the use of IDL arrays is nonstandard to Java. The typedef keyword would define an array of a particular data type, using a new data type name and the array length.

• IDL includes keywords in, out and inout to describe method arguments. A variable declared as in has a copy of itself passed to the called method. An out variable must be a reference to an object containing another object where the contained object may be substituted with another object
and the change is visible to the client (mimicking call-by-reference). A variable declared as `in-out` uses both semantics.

- Every time a `struct` or `interface` is declared, the IDL compiler generates an associated `Holder` class for use when the `struct` or `interface` is an `out` variable.
- All of the Java primitives have `Holder` classes available for them in the `org.omg.CORBA` package, and the IDL generates an `Operations` interface defining the `Holder` object needed.
- Only an `interface` can have `attribute` declarations.
- If the `attribute` keyword is used alone the compiler generates two methods—an accessor (`get` method) and a mutator (`set` method).
- Using the `readonly` keyword with `attribute` would generate only the accessor.
- In IDL, the `const` keyword declares a constant.
- Proxies are stand-ins for other objects. They allow a client to believe it is sending a message to one object when the client is really sending the message to another object.
- CORBA defines two related proxies—a stub and a skeleton. A stub is the client-side proxy and a skeleton is the server-side proxy. Both proxies hide the use of the ORB from the client and the server.
- In CORBA 3, static invocation choices can be synchronous (normal or `oneway`) or asynchronous (callback or polling).
- The use of a callback or polling mechanism is purely one of perspective—a callback is an invocation made from the server while polling is an invocation made from the client. The distinction has to do with a met condition (a server alerts a callback object) or an expected condition (a client poll an object to check on state changes).
- A normal synchronous call is a standard method invocation—a client invokes a method and blocks until the call completes.
- If an IDL method signature includes the `oneway` keyword, the compiler generates code that does not block on the invocation and returns based on the QoS settings.
- The CORBA Asynchronous Method Invocation specification supports both callback and polling models.
- The callback model supports (with various QoS options) the ability of a servant to call a client at arbitrary times at the discretion of the servant.
- In the polling model the client decides when to retrieve a possible result based on a call to a `oneway` method.
- When `structs` are instantiated in Java, all of the instance variables are `public`.
- Another way for a client to find a service is to use a stringified version of the service’s Interoperable Object Reference.
- The stringified IOR is valid at all times, as long as a server is available to handle any invocations.
- The `<data type name>` `Helper` classes are for the direct manipulation of mechanisms like the ORB’s object stream.
- The `<data type name>` `Holder` classes are for `out` and `inout` variable use as dictated in the code generated by the IDL compiler.
- Every call to a distributed object can throw an exception.
- Developers can find the exceptions thrown by a CORBA object by looking in the object’s associated IDL file.
• The keyword `raises` maps to the Java keyword `throws`.
• IDL `exceptions` map indirectly to Java exceptions inheriting from `java.lang.Exception` (they directly inherit from `UserException`).
• Standard CORBA exceptions map to Java exceptions as `final` classes.
• If a CORBA exception inherits from `SystemException`, then it is also a `RuntimeException`.
• The reason to send an object by value is the same as the reason to use `structs`—a local copy of an object has less access overhead than a remote object that needs repeated access.
• The target language of the mapped `struct` adds some overhead (in a nonobject-oriented target language, the overhead is almost nonexistent), but beyond that, the ORB adds nothing.
• Structurally, all IDL `structs` mapped to Java classes implement `org.omg.CORBA.portable.IDLEntity`, but do not inherit behavior or structure from any classes (except for the ubiquitous `java.lang.Object`).
• The Objects-by-Value (OBV) specification defines a new kind of `interface` type—a `struct` with behavior semantics (the ability to support the declaration of operations) called a `valuetype`.
• The advantage of `valuetypes` is the ability to pass copies of “objects” around a distributed system that contain not just data, but behavior.
• In the beginning, RMI required both stubs and skeletons to communicate, but, in JDK 1.2, RMI switched over to a skeleton-optional implementation protocol.
• CORBA specifies a Notification Service that delivers messages synchronously and asynchronously.

**TERMINOLOGY**

`interface`
Interface Definition Language (IDL)
Internet Inter-ORB Protocol (IIOP)
Interoperable Object Reference (IOR)
invocation transparency
location transparency
marshaling
`module`
Naming Service
object activation
Object Management Architecture (OMA)
Object Management Group (OMG)
Object Request Broker (ORB)
objects-by-reference
objects-by-value
`oneway`
opaque network reference
`out`
Portable Object Adapter (POA)
proxy
Quality of Service (QoS)
`raises`
`readonly`

activation
Asynchronous Method Invocation (AMI)
attribute
Basic Object Adapter (BOA)
class
Common Object Request Broker
`const`
CORBA Component Model (CCM)
CORBAFacilities
corbaservices
distributed computing
Dynamic Invocation Interface (DII)
exception
factory
General Inter-ORB Protocol (GIOP)
Helper
Holder
idlj
IIOP (Internet Inter-ORB Protocol)
implementation transparency
in
`inout`
SELF-REVIEW EXERCISES

26.1 Fill in the blanks in each of the following statements:
   a) ________ is the highest-level scope in an IDL file.
   b) In an IDL file, declarations of data without operations appear in a ________.
   c) An IDL interface maps to a Java ________.
   d) An IDL operation uses the keyword ________ to declare its use of an exception.
   e) A client would use the ________ service to retrieve a server’s IOR.
   f) The CORBA proxy used by a client is called a ________, and the proxy used by a server is called a ________.
   g) To find the initial object reference to standard services like the Naming Service, use the ORB method ________.

26.2 State whether each of the following is true or false. If false, explain why.
   a) An IDL interface maps to a Java interface of the same name.  
   b) An IDL struct can declare operations.
   c) An IDL valuetype can declare operations.
   d) The only way a client can get an object reference to a server is by using a Naming Service.
   e) IDL is an implementation language.

ANSWERS TO SELF-REVIEW EXERCISES

26.1  a) module.  b) struct.  c) interface.  d) raises.  e) Naming.  f) stub, skeleton.
     g) resolve_initial_reference( String serviceName ).

26.2  a) True.
     b) False. An IDL struct only contains data definitions.
     c) True.
     d) False. A client can read an object reference from a stringified IOR.
     e) False. IDL defines data, structures of data and interfaces to distributed objects, not their implementation.

EXERCISES

26.3  Write the IDL for a server named Server with one operation called getString and have the operation return a string. Implement the server. Write a client named Client that requests the string and displays the returned value.

26.4  Modify Exercise 26.3 to do the following:
     a) Return a struct containing the string.
     b) Change the get operation of the server. Have the client give the server the struct.
     c) Modify part a) above to have the client modify the struct object given to it by the server and have the client send the object back to the server.
d) Wrap the creation of the server (including the code to resolve the object reference) inside a client-side proxy. The client-side server proxy constructor should not throw any CORBA exceptions.

e) Wrap the `struct` in an object created in the proxy implemented in part d), using a factory method.

### 26.5
Create a service that returns object references to a `Customer` (do not make the `Customer` a `struct`). This service is the implementation of the factory design pattern as it creates and returns objects of a given type without revealing their actual type. Give the `Customer` two attributes and have a test client display the attributes (a primary key and an attribute).

### 26.6
Write a server with an operation that returns an IDL `unsigned short`. Have the server return `Short.MAX_VALUE + 1`. Display the value returned by the server in a `JOptionPane`. How does Java’s lack of support for `unsigned` values affect its interaction with servers that return values larger than Java can handle for certain data types?

### 26.7
Write a class named `FileServer` that returns a file requested by a client (use an array of IDL type `octet`). The client can display the file in a `JTextArea`. What limitations are there in the use of an IDL array?

### 26.8
Modify the `FileServer` to return a `FileStream` object. `FileStream` should cache the file on the server side until the client makes a request. `FileStream` should have an operation to return one line at a time.

### WORKS CITED


10. Siegel, 377.


12. Siegel, 164.


**BIBLIOGRAPHY**


Common Object Request Broker Architecture (CORBA): Part 2

Objectives

• To introduce the Dynamic Invocation Interface (DII).
• To understand the differences among BOAs, POAs and TIEs.
• To introduce CORBAservices, including Naming, Security, Object Transaction and Persistent State services.
• To understand the differences between RMI and CORBA.
• To introduce RMI-IIOP for integrating RMI with CORBA.

Two elements are needed to form a truth—a fact and an abstraction.
Remy de Gourmont

If two friends ask you to judge a dispute, don’t accept, because you will lose one friend; on the other hand, if two strangers come with the same request, accept because you will gain one friend.
Saint Augustine

The profoundest thought or passion sleeps as in a mine, until an equal mind and heart finds and publishes it.
Ralph Waldo Emerson
27.1 Introduction

The previous chapter introduced the world of distributed systems through the eyes of the OMG (Object Management Group). This is a group vision (the OMG has approximately 800 members at last count), and the architectural issues are all relevant and practical—these are vendors who have been building (and in some cases rebuilding) systems of this kind for years. Their combined expertise and resolve have produced far-reaching, influential specifications that have made huge enterprise-level systems possible. Their goal of an object marketplace is coming closer.

JavaIDL is the first step into the world of CORBA and distributed systems. Understanding of CORBA concepts at both the low level (i.e., object adapters) and high level (CORBA Services and CORBA Components) is important.
27.2 Static Invocation Interface (SII), Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)

There are two ways to invoke a request in CORBA—statically and dynamically. All of the examples presented up to this point use the Static Invocation Interface or SII.

SII relies on the compile-time definition of an object type (an IDL interface) and the object operations (IDL methods within an IDL interface). When the IDL compiler generates the client-side stub and server-side skeleton, there is no question what the contract is on both sides of the invocation. The generated stub/skeleton code defines the static type of the interface in an IDL file. From the server’s perspective, the server receives a request and, using any incoming parameters, executes code and returns any values necessary. The server-side ORB receives the pieces needed to complete a request, but does not know about the process that created the request.

The server-side skeleton’s ability to receive an invocation request that is independent of the mechanism to create the request makes a dynamically typed call to a server possible. A client has two choices available to make an invocation on a server: use the static stubs generated by the IDL compiler or manually create an invocation request programmatically. A client uses the Dynamic Invocation Interface (DII) API create and send an execution request directly to the server-side ORB without the assistance of a stub. The code written by a developer using the DII API looks similar to the code generated for the stub; the difference is the additional flexibility of programmatic control over the API instead of the hard-coded instructions found in the stubs.

On the other side of the invocation, the server can process the request in one of two ways: the static skeleton generated by the IDL compiler can process the incoming request or the servant (the object implementation under the control of an object adapter) can process the request manually. When a servant receives an invocation and manually extracts the data needed from the incoming request the servant uses the Dynamic Skeleton Interface (DSI). Clients and servers can use DII and DSI together or separately. Clients using the invocation interface can invoke operations on a server regardless of that server’s use of static or dynamic skeletons. Servers can accept incoming invocation requests from clients regardless of a client’s use of static stubs or dynamic invocations. Developers are responsible for passing and guaranteeing the proper type and number of arguments to the invocation.

The Interface Repository (IR) contains descriptive information about distributed objects. Such information includes the modules available, the interfaces defined within the modules, the names of the operations defined within the interfaces, the argument types acceptable to the operations, the return types of the operations and any exceptions raised. A client can discover this information about an object by retrieving metadata from the Interface Repository.

JavaIDL does not ship with an Interface Repository, so a truly dynamic working DII example is not possible with the ORB shipped with JDK 1.3. A basic code example is possible—we can bind directly to a remote object and, using a known operation signature, make a dynamic invocation to the object’s operation.

The steps needed to make a DII call (without using the Interface Repository) are as follows:

1. Obtain an object reference to the server object.
2. Create and initialize a Request object.
3. Invoke the `Request` and wait for the call to unblock (return).
4. Get the results.

Using the Interface Repository entails finding the Interface Repository entry for the object type in question after getting the object’s reference to the server object. The following are the new steps:

1. Obtain an object reference to the server object.
2. Look up the desired method in the Interface Repository.
3. Build the argument list for the target operation using the Interface Repository’s `OperationDef`.
4. Create and initialize a `Request` object.
5. Invoke the `Request` and wait for the call to unblock (return).
6. Get the results.

With the level of control possible using DII comes the high price of code complexity. Developers must craft every method invocation carefully, and the amount of custom code varies based on the complexity of the arguments, result types and exceptions. However, developers often find that the control that DII offers makes learning DII worthwhile. System-framework architects find DII useful in implementing hot-swappable components, administrative plugins and more. The Interface Repository becomes the central data store for finding registered object types—CORBA vendors are responsible for implementing robust, multiuser repositories that support the high-bandwidth, global distributed systems being developed today.

We have modified the `SystemClock` example to support DII. The files needed for `SystemClock` are `clock.idl`, `SystemClock.java`, `SystemClockOperations.java`, `_SystemClockImplBase.java`, `SystemClockImpl.java` and `SystemClockClient.java`. The only generated files needed are the server-side files, so `idlj` needs just the `-fserver` option set instead of `-fall`. The `Helper` and `Holder` files are unnecessary. `SystemClockClient.java` (Fig. 27.1) is the only file that needs updating.

```
// SystemClockClient.java
// Client that uses DII to request the system time from a servant.
package com.deitel.advjhtp1.idl.dii;

// Java core packages
import java.text.SimpleDateFormat;
import java.util.*;

// Java extension packages
import javax.swing.JOptionPane;

// OMG CORBA packages
import org.omg.CORBA.ORB;
import org.omg.CosNaming.*;
```

Fig. 27.1 `SystemClockClient` modified to support DII (part 1 of 3).
public class SystemClockClient implements Runnable {

    // object reference to desired server
    private org.omg.CORBA.Object timeServer;
    private ORB orb;

    // initialize client
    public SystemClockClient( String[] params ) throws Exception {
        connectToTimeServer( params );
        startTimer();
    }

    // use NameService to connect to time server
    private void connectToTimeServer( String[] params )
            throws org.omg.CORBA.ORBPackage.InvalidName,
                   org.omg.CosNaming.NamingContextPackage.InvalidName,
                   NotFound, CannotProceed {
        // Connect to the SystemClock server
        orb = ORB.init( params, null );

        org.omg.CORBA.Object corbaObject =
                orb.resolve_initial_references( "NameService" );
        NamingContext naming =
                NamingContextHelper.narrow( corbaObject );

        // Resolve the object reference in naming
        NameComponent nameComponent =
                new NameComponent( "TimeServer", "" );
        NameComponent path[] = { nameComponent };
        timeServer = naming.resolve( path );
    }

    // start timer thread
    private void startTimer() {
        Thread thread = new Thread( this );
        thread.start();
    }

    // talk to server on a regular basis and display the results
    public void run() {
        long time = 0;
        Date date = null;
        DateFormat format =
                DateFormat.getTimeInstance( DateFormat.LONG );
        String timeString = null;
        int response = 0;

        Fig. 27.1 SystemClockClient modified to support DII (part 2 of 3).
First, we need two instance variables to use in the client. Line 20 declares a reference to a generic CORBA object called `timeServer`, which will hold the reference to the remote server. The `orb` variable (line 22) holds the ORB reference used in both methods `connectToTimeServer` (lines 31–49) and `run` (lines 59–93) to connect the client to the server and to create support objects for the invocation to the server. Method `connectToTimeServer` places the newly created ORB object in variable `orb` and binding the client to server `timeServer`.
The invocation of server method `currentTimeMillis` occurs at line 78 (the original code is commented out at line 77). Prior to the `while` loop, the client asked the `CORBA.Object` returned by the call to `resolve` (saved in variable `timeServer`) to create and return a `Request` object. This `CORBA.Object` (the client-side construct that holds the server object reference) returns a `Request` object that the client uses to call the server. The client’s sole task is to set the `request` return type to an IDL `long long` (`TCKind.tk_longlong`). The call to `currentTimeMillis` does not need any incoming arguments, so the preparation of the dynamic call is complete once the return type of the method is set in the `request` (lines 70–71).

The call to `currentTimeMillis` becomes a multiline dynamic call to the target method of the `timeServer` (line 78 invokes the method and line 81 unmarshals the return value). The call to `invoke` sends the `request` object to our bound server and blocks until the operation returns. The client queries the `request` object for the `result`, queries the `result` for the `result`’s internal `value` and requests the value of the `result` as an IDL `long long` (line 81).

Running the transient Naming Service (`tnameserv`), `SystemClockImpl` and `SystemClockClient` should produce the same results as the `SystemClock` example in Chapter 26 with minimal additional overhead. The indirect call to `currentTimeMillis` did not receive any arguments, and the example does not need to reconstruct the `Request` object repeatedly to invoke the method on the remote object.

DII is composed of a number of useful technologies. Using the Interface Repository enables flexibility at runtime to discover more about the types available in a distributed system and yields options for a client to interact with future distributed objects. Think of the Interface Repository (and the API to access it) as a distributed version of the Java reflection mechanism. Highly-flexible frameworks can take advantage of the information contained within the IR as well as interested clients who need type information about a particular CORBA object they are working with at a particular moment in time.

JavaIDL is not a complete implementation of CORBA. The Javadocs for package `org.omg.CORBA` list the various parts of the JavaIDL packages defined in the API but not implemented in JavaIDL. Rather than work against Sun’s implementation, we recommend using a commercial ORB vendor’s CORBA implementation when the modeling and analysis of a Java-based distributed system are complete. Open-source implementations of CORBA for Java are also available that include a working Interface Repository.

### 27.3 BOAs, POAs and TIEs

When a client interacts with a distributed object, more than just a pair of ORBs are involved. When we consider implementation in a CORBA system, clients are made thin purposefully. Distributed systems are weighed heavily on the server, while client-side invocation issues are taken care of by the IDL generated stubs. Clients do not worry about issues such as load balancing or persistence—these are server-side issues. The indirection that enables this lopsided view is the *object adapter*, which stands between a distributed object and its ORB (Fig. 26.8). Many ORB services of use to a distributed object are accessed through its object adapter.¹ Some of the ORB services the object adapter transparently performs are IOR generation, security and activation/deactivation. The first adapter type specified by the OMG was the *Basic Object Adapter* or BOA. In CORBA 2.0, the definition of a BOA was
vague, so incongruities existed among various BOA implementations, which hindered the portability and interoperability of generated code that used ORBs from different vendors. To resolve these problems, the OMG deprecated the use of BOAs in favor of the POA, the Portable Object Adapter. The indirection introduced with POAs opens up many more possibilities and makes interoperability easier to guarantee. From one perspective, BOAs are easier to use (even if their behavior is inconsistent), but POAs are more widely employed.

A POA connects an object reference to developer-written code using the code found in the skeleton generated by the IDL compiler. As configurable objects, POAs allow fine-grained control over the object implementation (called a servant when attached to a POA). In the same fashion that our server inherited from the ImplBase definition (Fig. 26.4, line 11), a servant inherits from the POA base definition generated by the IDL compiler. In this case, our distributed object inherits the structure needed to be an object usable by a POA, and the POA controls all access to the servant based on various policies set within the POA. Policies describe characteristics of the objects under the control of a particular POA. These policies, once set, are immutable.

Three POA policies are ImplicitObjectActivation, IDAssignment-Policy, and RequestProcessingPolicy (there are seven policy types all together). The POA ImplicitObjectActivation policy of NO_IMPLICIT_ACTIVATION tells the POA an outside object created the servant under the POA’s control and passed the object to the POA using method activate_object or activate_object_with_id. An object id is a unique identifier that can be defined by either the developer or the system. Each object id is associated with a particular servant. Activating an object within a POA makes it available for use by clients. Deactivating an object makes the object unavailable, although the object may be activated again at a later time. The use of method activate_object or activate_object_with_id rests on the use of POA policy IDAssignmentPolicy, which can be set to either USER_ID or SYSTEM_ID. A POA policy of IDAssignmentPolicy USER_ID implies that a caller needing to associate a servant with a POA will define the servant’s unique identifier rather than letting the POA generate the object id (which would be the case if IDAssignmentPolicy SYSTEM_ID were used). Together with RequestProcessing-Policy USE_DEFAULT_SERVANT (and method set_servant) the above mentioned policies will configure a POA that will not create servants (however, the POA will route invocations to them), associate user-supplied object ids with various servants and route all invocations to a default servant if the incoming object id does not match an already active object. An example of this interaction is as follows: a client makes a request to a particular server using that server’s object reference. One of the pieces of information contained within an object reference is the servant’s object id. The POA, processing the invocation using the RequestProcessingPolicy, uses the object id either to find the servant that matches the object id or to invoke the default servant that uses the object id to perform a lookup in a database. If the default servant is called, it can create the requested object, returns the new object to the POA, and the POA would passes it on to the ORB for delivery to the client.

If there is no explicit need for an id, the POA can generate an id for internal use. In either case, when a request comes in for a particular servant, the POA uses a combination of policies and object ids to connect the request to a particular object. The indirection created by the use of a POA allows one POA to control one servant or many servants.
Although the combination of POA policies can be confusing, it enables various levels of QoS and scalability.

Another way for developers to use a POA is to wrap their servants in a TIE. TIEs enable interaction with a POA without having the servant’s object implementation inherit structure from a POAImpl. By delegating the CORBA functionality to the TIE object, the servant can inherit from some other base class freely (in Java, we want to inherit only when appropriate). A TIE object that mimics the servant’s API contains the created servant and invokes the proper operation at runtime.

**Good Programming Practice 27.1**

When using POAs, consider using TIE POAs. The potential distributed object can inherit from the Operations interface and gain CORBA functionality in a transparent fashion from the TIE POA wrapper class.

BOAs, POAs and TIEs rely on the code generated by the IDL compiler. The JDK-supplied IDL compiler, idlj, does not yet support the generation of POAs and TIEs, so developers are recommended to seek out 3rd party CORBA implementations if their system design warrants the use of POAs or TIEs.

### 27.4 CORBA Services

CORBA services are the Object Services of the Object Management Architecture (Fig. 26.7). CORBA services define base services and a support structure useful to a wide range of applications. The 16 services mentioned in Section 26.4 offer a wide range of functionality. The five services described in this section are the most commonly used (and are already the specified services for the CORBA equivalent of Enterprise JavaBeans, the CORBA Components Model, or CCM).

#### 27.4.1 Naming Service

The Naming Service associates name objects with an arbitrary value (known as name bindings). Any arbitrary value can be used, as long as it inherits from org.omg.CORBA.Object. In the context of a CORBA system, the association is between a name and an object reference. However, there are no restrictions as to the use of a Naming Service nor that the collection of objects the Naming Service contains must be homogeneous. A path to a name binding consists of zero or more naming contexts (a collection of unique name bindings). Resolving a name binding from within the Naming Service returns the object associated with the name. Binding a name to the Naming Service creates the association between a name and an object. Associating names to naming contexts creates a naming graph that represents the various paths available to the nodes holding a name binding. Multiple name bindings can point to a single object.

Naming represents a fundamental service in CORBA. Some of the CORBA services inherit IDL interfaces from other CORBA services, both for backward compatibility and for consistency of their APIs. Naming does not rely on other IDL interfaces, nor does the specification dictate administration of the service or what the name bindings mean. Anything that a client can associate with a name can be contained in the Naming Service. In addition, a group of Naming Services can participate as part of a naming graph to support the distributed discovery of name bindings.
27.4.2 Security Service

The Security Service is one of the more complex CORBA services. It consists of two levels. Level 1 provides basic security for:

1. User authentication.
2. Invocation security.
3. Availability of authentication principals to security-aware applications.

Level 1 allows applications to ignore the system-security requirements. The security needs are limited in scope, and the ORB will usually handle these needs to the total ignorance of the objects involved in an invocation. A normal ORB does not care about additional processing of object invocations. However, for security to work pervasively, a security-enabled ORB must support the passing of credentials from an initial caller (and any intermediate objects) to a target object.

Level 1 security requires support for the no delegation and the simple delegation models among a client, an invoked object and a final target object. In the no delegation model, a client passes credentials to an intermediate object—this object creates intermediate credentials for use in calling the final target object. In the simple delegation model, an intermediate object can use the client’s credentials to make invocations on a target object. If there are no restrictions on how an intermediate object can call the target using the client’s credentials, the intermediate object impersonates the client.

Level 2 is everything Level 1 provides plus the following security areas:

1. More fine-grained user authentication (i.e., user authentication can be checked at method invocation).
2. Greater invocation security.
3. Auditing.
4. Finer control over secure invocations.
5. Delegation.
6. Administrators can set security policies.
7. Discovery of security policies by security-aware applications.
8. Discovery of security policies by ORBs and other services.

Security level 2 gives a system full use of all the functionality listed previously (through the use of APIs defined by the various Security Service interfaces). Separate administration programs modify security policies.

27.4.3 Object Transaction Service

The Object Transaction Service (OTS) enables CORBA objects to execute as parts of distributed transactions. The OTS specification was one of the first of the CORBA services specifications and has undergone recent changes to increase its flexibility and to fix various implementability issues. A transaction describes a collection of interactions where multiple users may access and/or modify data and the data’s integrity is guaranteed (a transaction is also known in the database world as a unit of work). The acronym ACID describes the four standard requirements for reliable transactions:
• Atomic—the completion of the transaction signals either full success or complete failure; if five steps were to be completed within the course of a transaction, then all five steps must terminate successfully or the changes made in each step must be undone. In an atomic transaction, there will never be a situation where some of the steps completed, but others did not.

• Consistent—the effects of the transaction are repeatable and predictable. Executing the same steps on the same data set will always return the same result (also known as preserving invariants).

• Isolated—the transaction is not interruptible from the outside and gives no indication if execution is proceeding serially or concurrently.

• Durable—short of a catastrophic failure (power outage, earthquake etc.), the transaction results are persistent.

A transaction completes in one of two ways—committed (the changes made persist) or rolled back (all changes made to the data are discarded). The OTS adds to the properties of a reliable transaction service the ability to control transactions across a distributed system.

Functionally, the Object Transaction Service supports flat and (optionally) nested transactions. Flat transactions are the most common type. Nested transactions support ACID for the duration of a child transaction, yet allow for a partial rollback if a child transaction fails. A complete rollback of child transactions occurs when the parent transaction fails. When the top-level transaction—the transaction with no parent—fails, the entire transaction rolls back.

The OTS allows developers to add distributed transactions to existing systems. CORBA facilitates the joining of heterogeneous systems. The OTS, in conjunction with standards such as the Open Group’s XA Specification, enables a client to use transactions implicitly or explicitly. An implicit use of a transaction sends the transaction context transparently from ORB to ORB once the transaction is created. An explicit transaction passes the transaction context as a parameter to any called methods. OTS also allows servers to register with the transaction service. To enable the joining of heterogeneous systems, the OTS also supports the X/Open Distributed Transaction Processing model, which allows OTS to communicate with with procedural transaction systems.

In CORBA 3, the Asynchronous Method Invocation (AMI) specification dictates the behavior of asynchronous calls in a distributed system. In a CORBA system the behavior of asynchronous invocations is almost purely a client-side issue; servants have their transactional requirements set through their POA, and clients making asynchronous calls must have their invocations handled by intermediate routers that will create a new transaction context involving the servant.

POAs dictate the transactions types supported by servants—allow or require only shared transactions, allow or require only unshared transactions, or allow or require either transaction type. If the transaction policy is set at the POA level, that means every object (servant) under the control of the POA has the same transaction policy. A POA can control more than one object at a time, so the objects under the POA’s control have the same transactional requirements.

The concepts of transactional clients, transactional objects and recoverable objects define the OTS. A Transactional Client interacts with the OTS to create and commit or rollback a transaction. A transactional object’s behavior varies when invoked within a transac-
tion, but object’s data may not be recoverable. A recoverable object is a transactional object in which data is recoverable (e.g., an object that represents a row in a database). Recoverable objects maintain their data and, in the event of a persistence failure, assist in restoring their lost state. Two types of application servers that use transactional objects are transactional and recoverable servers. A transactional server has no recoverable objects, whereas a recoverable server has at least one recoverable object. The OTS specifies that both server types can roll back a transaction, but only the recoverable server can participate in the commit.5

The Java Transaction Service (JTS) is the Java implementation of the distributed transaction service. The API of the JTS is defined by the Java Transaction API (JTA). The JTS uses the CORBA OTS specifications to define the protocol between transaction managers during the exchange of transaction contexts.

27.4.4 Persistent State Service

The Persistent State Service (PSS) stores and retrieves objects. In conjunction with an OTS, the PSS abstracts the interaction between objects and datastores. Ideally, systems should persist their objects into object datastores. However, relational databases dominate the marketplace—mapping objects into these databases in a consistent, predictable manner is not trivial. The Persistent State Service relies on the Object Transaction Service and Security Service to maintain transactional integrity and access control.

In the same way that IDL defines the interface of a distributed object, a new Persistent State Definition Language (PSDL) defines a distributed object schema in a portable fashion (PSDL is a superset of IDL). Two new constructs are storagetype and storagehome (usable with or without the keyword abstract). A PSDL file contains two definition types: an abstract definition and a concrete definition. In addition, a factory (or home) can be declared that creates objects of the newly defined type. The abstract storagetype or abstract storagehome do not define concrete object definitions—rather, they define the portable definition of the persistable state of a CORBA object. The concepts and constructs of an object factory and factory-created objects should look familiar to anyone working with Enterprise JavaBeans (Java interfaces EJBHome and EJBObject are the corresponding base classes used in the definition of EJBs). Figure 27.2 shows an example of PSDL for a customer storagehome and storagetype.

```plaintext
1 // The schema for a domain object. This is the abstract definition
2 // needed by the PSS. The concrete definition of Customer is below.
3 abstract storagetype Customer {
4     // The accountNumber is our primary key
5     readonly state string accountNumber;
6     state string name;
7 }
8
9 // The factory to be used to retrieve Customer objects
10 abstract storagehome CustomerHome of Customer {
11     // The creation method will create persistent Customers
12     Customer create( in string accountNumber );
13 }
```

Fig. 27.2 Persistent State Definition Language example (part 1 of 2).
Line 3 declares the abstract storagetype for the Customer. In addition, Lines 5–6 declare a read-only accountNumber and readable/writable name using the keyword state to notify the PSDL compiler of which fields to persist. The Customer is instantiated by a factory, so there should be a declaration of an abstract CustomerHome factory.

Lines 10–13 declare storagehome CustomerHome. This factory creates Customers using method create, each with an accountNumber as the primary key. A developer does not implement the home class. The PSS software generates the code needed to map the data store to the declared definition of the object. The PSS vendor is responsible for delivering a tool that maps these objects to the data store in which the objects persist.

The two abstract declarations (lines 3 and 10) and the two concrete declarations (lines 25 and 27) are enough information for the PSDL compiler to generate the code creating CustomerHomeImpl and CustomerImpl classes along with any needed supporting interfaces and abstract classes.

### 27.4.5 Event and Notification Services

The Event Service defines the mechanism that decouples the delivery of events (messages) from the source of the events. The Event Service is responsible for keeping track of such as ActionEvent and ActionListener for use by objects that want to send or receive events. In a similar fashion, EventChannels use the CORBA data type Any to send an event of any type (either primitive or object) across the network in a consistent fashion, as there are no predefined event types in the Event Service specification. However, there is no distributed Java event model, whereas the CORBA model is distributed.

In general, a supplier creates events that are processed by a consumer. In a push model, a supplier sends event messages asynchronously to all consumers registered to receive the messages. In a pull model, the consumer polls the supplier for events—if none are available, the consumer may block until events become available. A nonblocking consumer looking for events has to poll the supplier on a regular basis for events. For example, Java uses the push model for Swing and AWT events. A handler object implements the ActionListener interface and registers with a Swing component, which notifies the handler of state changes to the GUI component by calling the handler’s actionPerformed method with an
**ActionEvent.** The GUI component sends the event to the handler object asynchronously. However, Java events are different than Event Service events—the Event Service does not define an event type. Java events are strongly typed, whereas CORBA Event Service events can be typed or untyped, based on the kind of event channel used.

Suppliers and consumers can be either **pull or push,** and the type of model they use does not have to match. A push supplier can create messages for consumption by a pull consumer, and a push consumer can wait passively for messages to arrive from a pull supplier. The Event Service acts as the queue (also called the **channel**) where events wait for their consumers. The supplier can either push events into the event channel or have events pulled by the event channel. Once the event is sitting in the queue, the channel pushes the event to a push consumer or waits for the pull consumer to ask for the event. There is no mechanism for finding particular event channels and no defined quality of service in the Event Service, but these important capabilities are added by the Notification Service.

The **Notification Service** is the industrial-strength Event Service. The Notification Service is a direct extension of the Event Service, as it inherits from the original Event Service IDL interfaces. Objects can create and destroy event channels arbitrarily and can filter their output using **Filter Objects** and the **Object Constraint Language** grammar (originally defined for the Trader service and known as the Trader Constraint Language or TCL). The Notification Service provides full support for the Event Service. Method **EventServiceHelper.narrow** safely converts the object reference of a Notification Service into an Event Service object reference. In addition, the Notification Service also defines an event type named **StructuredEvent.**

Figure 27.3 depicts the levels of indirection inherent in the CORBA Event/Notification Service and shows how the decoupling of the event’s supplier from the event’s consumer adds flexibility as additional objects come into play. The standard flow of a **StructuredEvent** starts with the supplier finding the object reference to the Notification Service (an object of type **EventChannelFactory**). The **EventChannelFactory** creates an **EventChannel,** and the supplier asks the **EventChannel** to return a **SupplierAdmin** object. The **SupplierAdmin** returns one of many **Consumer** proxy types (such as **StructuredProxyPushConsumer**), and the supplier uses this consumer proxy. A **StructuredProxyPushConsumer** is a proxy to an event channel that accepts **StructuredEvent**s from a supplier. The **EventChannel** created for the **SupplierAdmin** created the **StructuredProxyPushConsumer,** so all the objects in this chain communicate on the same event channel. Using a non-blocking push method, the supplier can create and push **StructuredEvent**s into the channel and prepare to send more events as appropriate.

The consumer does the same from the opposite side. The consumer finds the reference to the Notification Service, **narrow**s the object reference to an **EventChannelFactory** reference, gets a known channel, retrieves a **ConsumerAdmin** object, retrieves a **StructuredProxyPushSupplier,** connects to the supplier and waits for events to arrive.

A concrete use of the Notification Service is a chat application. A chat object that sends a message is a supplier. A chat object that receives that message is the consumer. Traditionally, chat applications use a pure push model (the chat message supplier pushing messages to the chat message consumer waiting to be pushed). The application does not need to use custom IDL if the event had carried standard CORBA data types. If custom data types are needed (for example, **struct** or **valuetype TextMessage**), the developer has to
write the appropriate IDL. A chat application using the Notification Service is an example of a peer-to-peer application—an application in which clients communicate with each other directly, without going through a central server.

27.5 EJBs and CORBA components

Enterprise JavaBeans (EJBs) define “a standard component architecture for building distributed object-oriented business applications in the Java™ programming language.”\(^6\) The OMG released the CORBA Component Model (CCM) Request for Proposal (RFP), which recommended the JavaBeans component model as the basis of a server-side framework. During the analysis and design of a pure Java server-side component framework, Sun decided not to use JavaBeans as the basis for a server-side distributed component architecture—instead, Sun decided to use Enterprise JavaBeans (EJBs are JavaBeans in name alone—JavaBeans support the throwing of events while EJBs are not required to support events). The CORBA Component Model submitter companies chose to follow and based their component architecture on many of the ideas from EJB, while taking advantage of the advanced features already found in CORBA.

---

**Fig. 27.3** Supplier-to-consumer flow using the Event/Notification Service.
The OMG defines technology standards to support an object marketplace, thus facilitating the construction of object-oriented systems. Components, as coarse-level objects, bring this marketplace one step closer to reality. The use of encapsulation, inheritance, polymorphism and dynamic binding enables reusable architectural frameworks. The designers of CCM based CORBA components on the Component Implementation Framework (CIF) architecture. CIF specifies a successful architectural pattern based on CORBA and CORBA technologies. CIF defines a superset of the Persistent State Definition Language called Component IDL (CIDL) a component model for CORBA objects and a container-programming model where CORBA components exist at runtime.

The CCM defines several additions and extensions to CORBA to make CCM possible. An understanding of Enterprise JavaBeans facilitates understanding these concepts. The CCM specification extends IDL with the keywords listed in Fig. 27.4.

These keywords enable developers to define components in a high-level fashion, allow for single-class inheritance and multiple-interface inheritance (for those mapped languages that support an inheritance model) and support all the normal attribute and operation syntax found in IDL interfaces (including the recently added keywords private, public, attribute etc.).

The Component Interface Definition Language (CIDL) is a superset of the Persistent State Definition Language introduced in Section 27.4.4. As a superset of PSDL, CIDL defines components in a way that enables the automatic generation of the component's persistence code. CIDL also defines component implementation as well as state management. A developer compiles a .cidl file with a CIDL compiler.

When a collection of components is complete, the developer organizes the components as an assembly together with a descriptor that describes how the components are deployed. The descriptor is an extension of the Open Software Description (OSD) Format (an XML vocabulary). OSD describes the installation and distribution rules of software for the desktop. CCM defines a modified version of OSD to support component packaging.

The most interesting parts of CCM are its container programming model, how the POA fits into CCM and EJB interoperability. Commercial CCM implementations are expected any day, but for now, the OpenCCM Platform out of the Université des Sciences et Technologies de Lille (LIFL) is available. OpenCCM is a work in progress and is free.

A Container class creates a containment hierarchy grouping components and other containers together (such as a Panel from the Java AWT package). The Container programming model is a runtime environment in which component implementations use their enclosing containers to access various services the container provides. The containers use the implementation of services available to them by their deployment environment (an application server, for example). Four key areas make up the CCM container programming model:

---

**IDL keywords to support the CORBA Component Model (CCM)**

- component
- home
- provides
- consumes
- import
- setRaises

*Fig. 27.4* IDL keywords to support the CORBA Component Model (part 1 of 2).
External types—the interfaces seen by a client wanting to communicate with a component.

Container types—the API that the component uses to communicate with the runtime container.

Container Implementation types—different containers have different relationships with the surrounding system. The three CORBA containers types available are stateless, conversational and durable. Each container type defines its system support.

Component Category—where a component fits in the overall framework. The external types and container types define this view.

The three container implementation types are subtly different from one another. A stateless container expects to call the servant referenced by a POA regardless of the object id used to make the invocation. A conversational container expects a transient servant referenced with a particular object id to handle the operation. A durable container has a persistent servant that answers to a particular object ID to handle invocations. These container types directly reflect a subset of the policies used by POAs, because CCM containers can be viewed as specialized POAs. POAs are associated with containers and the POAs are configured based on their associated container type.

The container defines the API for component security, persistence, transactions, events and lifecycle (creation, persistence, etc.). Using a container’s internal interfaces, a component has full access to the services a container supports. The component implements callback interfaces to allow container-to-component communication. Containers can create either transient components or persistent components. A component is associated with only one container type (stateless, conversational or durable).

CCM containers support both single and multithreaded models (serial or concurrent access). A container with a policy of serialize forces all invocations on a component to be handled sequentially. A policy of multithreaded advises the container as to the component’s ability to handle multithreaded access to the component’s state. The deployment descriptor specifies the component’s threading policy.

Components can be either transient or persistent. Factories provide creation points for components, which may or may not have primary keys. Finder methods do not create objects; they simply find requested objects using some arbitrary key criteria. When a factory client requests an object to be found at runtime the object may be instantiated to complete the request. From the client’s perspective the object was found from an existing data store and did not need to be created. Logically, an object is only “created” when a
factory **create** method is invoked. CIDL defines factories and finders as the creation point for our components, but transient components can only be created by factories while persistent components can be created by factories and located by finders. Transient components do not have primary keys (or at least not primary keys available outside the container where the component exists), so finder methods have no key with which to search for them. Persistent components, having primary keys attached, support two kinds of persistence—container-managed and component-managed. *Container-managed persistence* is maintenance and administration intensive, while *component-managed persistence* is developer intensive. Figure 27.5 lists the CORBA component types and their descriptions. 8

*Activation* and *passivation* are actions around the invocation boundaries of an object operation. When a request comes in to an ORB to invoke an operation on an object, the ORB sends the request to the POA. The POA sends the request to the container managing a particular component, and the container activates the object. The object’s deployment information defines the passivation policy of the object. The passivation policies are *method* (passivate when the operation completes), *transaction* (passivate at the conclusion of the transaction), *component* (passivate at the request of the component) or *container* (passivate at the container’s discretion). Activation and passivation occur when a container calls a component using one of the component’s callback interfaces.

In distributed systems, clients do not create objects directly; they discover the objects needed. The discovery process can be either through a file containing the object’s IOR or through a Naming Service. Factories create our components, or, in CCM terminology, **ComponentHome** objects create our components. A component definition must also define the component’s factory and, if the component is persistent, a **find** method using the component’s primary key.

CORBA components use a subset of CORBA services for the Component Implementation Framework. These include security, transaction, persistence and notification services.

The Security Service defines authorization based on roles. Every component can have differing security requirements (defined in the component’s deployment descriptor), but the container is responsible for keeping them consistent. If the container updates the security of a component, the new policy stays in effect until the invocation of another component occurs and that component’s security policy becomes active. The standard Security Service specification did not change to support these requirements.

The CCM allows the use of a lite version of the Object Transaction Service. The **Persistent State Service** specification defines the lite version of OTS to allow the use of smaller implementations of the OTS (one which allows the underlying datastore to be responsible for all transactional access). The OTS interfaces do not change in any way, which means replacing the underlying OTS does not change code using the OTS. In addition, the container can control the transaction boundaries or the component can. An example transaction boundary is at the activation/passivation point.

The Persistent State Service manages object persistence. In container-managed persistence, the use of PSS is transparent to the component. This might seem like the most convenient method for persistence—container-managed persistence is not necessarily ideal for every design situation. Components managing their own persistence should still use the PSS to save their state, but this comes at the price of future maintenance.
CCM components access an event service indirectly based on whether or not they use the IDL keywords `publishes` and `emits`. The container in which the component runs mediates use of the event service by the component. The notification Quality-of-Service settings needed by a particular component are also set by the container. The notification and event services supported by the container have to support only the notification and event service subsets defined by the CCM specification, respectively. Using the generated component API does not preclude the direct use of the Notification Service, but component developers are encouraged to define the events in the IDL as a way of keeping the component’s functional description in one location.

The keywords `publishes` and `emits` are similar. The difference is in the number of consumers allowed to receive events sent by components. Any number of consumers can
subscribe to an event that a component `publishes`, whereas only one consumer can subscribe to an event a component `emits`. Events declared using `publishes` are expected to be part of the shared interface of the component (the API available to all clients). The event channel created for `emits` is expected to be a private channel used internally by the system. Figure 27.6 declares a `Customer` component that `publishes` a `PropertyEvent` to multiple consumers if the `Customer` has exceeded their credit limit and `emits` an `InvariantEvent` to one particular event consumer if the `Customer`’s internal state has been set improperly.

Components, like other CORBA objects, can communicate with CORBA services. Components use ORBs to discover available services. This discovery process implies the use of the Naming Service, but CCM introduces a `HomeFinder` service for components to discover `ComponentHome`s for other components. `HomeFinder` is a kind of Naming Service strictly for discovering `ComponentHome`s.

CCM defines a framework for implementing enterprise-level distributed objects using CORBA. The CCM infrastructure enables the use of a subset of CORBA services (Naming, Security, Transaction, Persistence and Event). The IDL-defined component factories subclass `ComponentHome`. These component factories are found using `HomeFinders`—this is another use of the Naming Service. CCM components come in four types—service, session, entity and process. Based on the amount of state the components declare and whether any of the state needs to persist, these components can delegate persistence to the container in which they exist (container-managed persistence) or can handle the persistence on their own (component-managed persistence). CCM containers can be single-threaded or multithreaded, based on whether the CCM component can handle multithreaded access. Clients access the components through the container in which the components execute.

Enterprise JavaBeans (Chapters 14 and 15) and CORBA components share a significant amount of lineage. A design point in the CCM specification is that CORBA components-programming model (they run in the process space of a runtime environment, either an application or EJB server). The EJB specification dictates that the runtime environment must support Naming, Security, Persistence and Transaction services. The EJB distributed-object framework uses RMI as the client-side/server-side glue, defines `EJBHome` as the base interface for component factories and defines `EJBObject` as the base interface for components. EJBs come in two types—`session` (stateless and stateful) and `entity` (stateful). Stateless session beans never persist, stateful session beans are persisted based on the container’s policies and entity beans always persist. EJBs should be single threaded; the containers are not specified as having configurable threading policies. Clients access EJBs through the EJB container.

```java
// Our Customer can send two events:
// creditEvent if the customer’s credit limit has been exceeded
// and internalStateEvent if the some data in the customer
// has not been updated properly

class Customer {
    publishes PropertyEvent creditEvent;
    emits InvariantEvent internalStateEvent;
}
```

Fig. 27.6 `Customer` component IDL definition demonstrating keywords `publishes` and `emits` for issuing events.
The CCM and EJB models are quite similar. The CCM specification defines two levels of components—*basic* and *extended*. A *basic* component mirrors the EJB model almost exactly. A basic component lives in a container that supports only single threading; the component uses only the security, transaction and persistence services; the component exposes only a single interface to clients. An *extended* component supports multiple interfaces. The CCM defines advanced storage types and allows them to be properly persisted. An extended component also supports the use of event services.

Sun developed Enterprise JavaBeans independently of the CORBA components architecture. However, the OMG and Sun worked closely in joining the two architectures during the design of CCM. Enterprise JavaBeans is now an official subset of the CCM architecture. Ideally, an EJB implementation built on top of a CCM system would be a much more extensible system than an EJB system built from scratch. The EJB Specification, Version 2.0 requires support for IIOP. Vendors who ship EJB 2.0-compliant products must support the CORBA protocol. IIOP support does not mean that EJB containers can handle CCM components. Developers can write CCM components in any language for which the CORBA CCM mappings exist. The implication is that EJB is the first real introduction of a distributed components architecture (not just distributed objects). Systems integration is entering a mature phase of development, in which COBOL components can interact side-by-side with Java components or C++ components in software frameworks that use similar concepts and APIs.

### 27.6 CORBA vs. RMI

CORBA is a comprehensive view of distributed systems architecture, whereas RMI describes only the communication proxies and protocols between a client object and server object. The OMA specification defines a road map for integrating systems that use object technology and system architecture concepts. CORBA follows the OMA specification and benefits from the breadth of industry expertise that created it. The member organizations of the OMG have evolved CORBA into a useful and robust technology.

#### 27.6.1 When to Use RMI

RMI is suitable for smaller distributed applications in which scalability, architecture, heterogeneity, and extensibility are not major concerns. For a time, there was no long-range architecture road map for RMI the way CORBA has the OMA. Architecturally, building the infrastructure for an RMI system is not a trivial undertaking. A mapping of RMI and RMI capabilities to the OMA encompasses only Applications Objects and the ORB. Applications Objects are specific to the application or system and therefore have no formal specification. The RMI communications infrastructure mimics an ORB (with the RMIREgistry used as a naming service). In addition to missing standard infrastructure—e.g., such as that defined by the CORBA Services and CORBA Facilities—RMI does not define Quality of Service or asynchronous invocations. RMI has no equivalent to the Portable Object Adapter and the range of control afforded through use of a POA. In addition, RMI automatic and dynamic class loading permits a client to perform some operations locally rather than via network calls. CORBA has the `struct` keyword, but this places the burden of implementation and binding on the client, which is not ideal for some applications.
Enterprise JavaBeans, on the other hand, is a larger view of systems and where technology such as Java is heading to address the concerns of customers with large-scale, real-world problems. Security and data manipulation are the highest risk elements of many systems, implying the need for specialized subsystems to mitigate those risks. These subsystems (i.e., security and persistence) and others (for example, transactions and messaging) appear repeatedly as standard players in nontrivial systems. Prior to CORBA, systems integrators built custom (sometimes brittle) frameworks to accomplish reliable communications between heterogeneous systems and standard services. Once CORBA and RMI standardized the communications infrastructure, frameworks began to grow around them. A framework such as Enterprise JavaBeans is a reliable and cost-effective way to build large-scale distributed systems. EJB is the Java implementation of a first-generation framework with direct lineage to CORBA.

RMI allows objects to participate in distributed systems consistently and predictably. RMI is a natural choice for distributed systems built in Java. RMI is inappropriate for system architectures that require container services, load balancing or other framework functionality without custom development. A pure Java-distributed system needs to be considered from an architectural perspective where the distributed issues can be discovered and resolved before the implementation issues are decided. Too often, the question of where to use RMI becomes the issue, when a more appropriate question is “what design problems are best solved using RMI?”

### 27.6.2 When to Use CORBA

If developers should not use custom RMI except for small projects, this leaves IDL in an awkward position. IDL is not difficult to learn, but it is not a trivial undertaking for developers who are unfamiliar with C/C++ syntax. CORBA can have a steep learning curve for those starting to learn object technology. Yet CORBA is still an excellent way to develop the full range of distributed systems, even without a framework like CORBA components. The reason CORBA is a good choice in the design of nontrivial systems has to do with implementation concepts (e.g., OMA). In the area of connectivity, both RMI and CORBA share the same protocol (IIOP), but RMI falls short in many other areas compared to CORBA.

- **Architecture:** RMI does not define an architecture in any way, shape or form. CORBA has defined an established distributed-systems architecture since the 1980s. Moving to distributed component architectures similar to Enterprise JavaBeans had always been in the road map. Proven architecture, like EJB (which uses mechanisms such as RMI), helps the OMG continue standardizing accepted/existing practice.

- **Quality of Service (QoS):** The Quality-of-Service specification establishes policies for distributed-object behavior at many levels. There are no programmatic QoS concepts in Java, RMI or EJB yet.

- **Scalability:** RMI supports the concept of an object adapter but only as a basic mechanism to hide the method-invocation complexity. The concept behind the Portable Object Adapter, which has various selectable policies for activation (including one that allows a single servant to handle calls to different object types) is unknown in RMI.
• Heterogeneity: Java, as an implementation language, runs on many different operating systems and, by extension, several kinds of hardware platforms. This differs from CORBA’s view of heterogeneity—CORBA enables interoperability between systems written in different languages and different operating systems and hardware platforms. CORBA has no implementation except that developed by various vendors. CORBA encourages vendors to port CORBA’s technology to many platforms and languages.

• Extensibility: Being hardware-neutral, language-neutral, and location-transparent keeps CORBA in the forefront of many system-integration projects.

27.6.3 RMI-IIOP

In a joint effort, Sun and IBM implemented *RMI-over-IIOP (RMI-IIOP)* to replace RMI’s underlying communication protocol (Java Remote Method Protocol or JRMP) with CORBA’s IIOP. Inprise, Netscape, and Oracle, working with Sun and IBM, specified a reverse mapping of Java-to-IDL to allow the RMI compiler (**rmic**) to output IDL, stubs and skeletons needed by a server to accept invocations and a client to make the invocations. Restrictions in the reverse mapping include the following:

1. Constants can be of primitive types or of class **java.lang.String** only.
2. IDL normally does not support overloading of method names.
3. A class cannot inherit a method with the same signature from two different interfaces.
4. Interfaces and value types must be **public**.
5. The compiler considers packages and interface names to be identical if the only difference is case sensitivity.

The mapping also identifies four runtime limitations:

1. Sending a tree graph (an object containing references to other objects) from ORB to ORB may be problematic if multiple nodes point to one object (copies will probably be sent instead).
2. CORBA does not define distributed garbage collection.
3. Casting stubs may not work properly, so using the static method **narrow** of class **java.rmi.PortableRemoteObject** is encouraged. RMI downloads the stubs needed by the client to communicate with a server, but CORBA does not support that behavior.

Writing a distributed application using RMI-IIOP involves the standard RMI development steps (Chapter 13, Remote Method Invocation) with a small number of changes:

1. Use **javax.rmi.PortableRemoteObject** instead of using **java.rmi.UnicastRemoteObject**. A workaround would be to use value-types (discussed in Chapter 26).
2. Use the **JNDI (Java Naming and Directory Interface)** instead of the RMI Registry.
3. Do not downcast remote objects to subclass types; use method **narrow** of class **PortableRemoteObject** to cast distributed objects as subclass types.
27.7 RMIMessenger Case Study Ported to RMI-IIOP

In this section, we modify the RMI messenger program of Chapter 13 to use RMI-IIOP. Porting the RMI messenger to RMI-IIOP is easier than porting the application to CORBA (Chapter 26). The remote interfaces (ChatServer, StoppableChatServer and ChatClient) remain the same. The modular approach we used to design the Deitel messenger system enables us to reuse the supporting classes and interfaces (ChatMessage, ClientGUI, MessageListener, MessageManager and DisconnectListener) also without modification.

For the RMI-IIOP messenger, we provide new implementations of the ChatServer remote interface (ChatServerImpl), the ChatServerAdministrator, the MessageManager interface (RMIIIOPMessageManager) and the client launcher (DeitelMessenger).

27.7.1 ChatServer RMI-IIOP Implementation

Class ChatServerImpl (Fig. 27.7) implements the ChatServer remote interface as a subclass of class javax.rmi.PortableRemoteObject (line 20), which is the base class for RMI-IIOP remote objects. No changes are required in either the ChatServer or StoppableChatServer remote interfaces, which class ChatServerImpl implements (line 21). Class ChatServerImpl does not implement method register, which registered the Activatable RMI ChatServer with the RMI registry. Instead, the RMI-IIOP version of class ChatServerAdministrator (Fig. 27.8) handles registration with name services. The remainder of the RMI-IIOP ChatServerImpl is identical to its RMI counterpart.

Class ChatServerAdministrator (Fig. 27.8) is a utility program for starting and stopping the RMI-IIOP ChatServer implementation. Method startServer (lines 21–53) creates an instance of ChatServerImpl (line 27) and registers this ChatServer with a naming service (lines 30–35). The RMI-IIOP equivalent to the RMI registry is the CORBA Naming Service. Instead of connecting to a naming service in a CORBA-specific way, Java recommends using the Java Naming and Directory Interface (JNDI), which abstracts the concept of naming and directory services. Package javax.naming (line 13) contains JNDI classes and interfaces. Line 30 creates an InitialContext, which represents the naming service. Line 35 invokes method rebind of interface Context to bind the ChatServerImpl object to the name "ChatServer" in the naming service, thus making the ChatServer available to clients. At runtime, the InitialContext uses two system properties to discover the underlying naming service (the Javadocs for javax.naming.InitialContext and javax.naming.Context specify the system-naming properties and useful default values). Property java.naming.factory.initial defines the class name of the factory that will create the InitialContext object. Class com.sun.jndi.cosnaming.CNCtxFactory is the default factory class for connecting to CORBA Naming Services. Property java.naming.provider.url specifies the URL for the Naming Service, including the port to which the Naming Service is attached (e.g., iiop://localhost:1050).

Method terminateServer (lines 56–91) locates and stops the running ChatServer and removes the ChatServer registration from the naming service. Line 62
creates an InitialContext for the naming service in which the ChatServer is registered. Lines 65–66 invoke method lookup, passing as an argument the name under which the ChatServer is registered. Method lookup returns an Object reference to the ChatServer remote object. Lines 69–71 create a StoppableChatServer reference to the ChatServer remote object. Recall that RMI-IIOP requires that remote objects be converted to specific types using method narrow of class PortableRemoteObject, as opposed to using casting. Method narrow takes as arguments an Object reference to the remote object and the Class object for the target reference type. Line 71 passes remoteObject and the Class object for interface StoppableChatServer to method narrow. Line 74 invokes method stopServer of interface StoppableChatServer to terminate the ChatServer. Line 77 invokes method unbind of interface Context to remove the ChatServer's registration from the naming service. Lines 82–84 catch a NamingException if the ChatServer is not found in the naming service. Lines 87–89 catch a RemoteException if there is a problem obtaining the ChatServer remote object or invoking its remote methods.

```java
public class ChatServerImpl extends PortableRemoteObject
    implements ChatServer, StoppableChatServer {

    // Set of ChatClient references
    private Set clients = new HashSet();

    // construct new ChatServerImpl
    public ChatServerImpl() throws RemoteException {
        super();
    }
```

Fig. 27.7 ChatServerImpl implements the Deitel messenger ChatServer using RMI-IIOP (part 1 of 3).
```java
// register new ChatClient with ChatServer
public void registerClient( ChatClient client )
    throws RemoteException
{
    // add client to Set of registered clients
    synchronized( clients ) {
        clients.add( client );
    }

    System.out.println( "Registered Client: " + client );
} // end method registerClient

// unregister client with ChatServer
public void unregisterClient( ChatClient client )
    throws RemoteException
{
    // remove client from Set of registered clients
    synchronized( clients ) {
        clients.remove( client );
    }

    System.out.println( "Unregistered Client: " + client );
} // end method unregisterClient

// post new message to ChatServer
public void postMessage( ChatMessage message )
    throws RemoteException
{
    Iterator iterator = null;

    // get Iterator for Set of registered clients
    synchronized( clients ) {
        iterator = new HashSet( clients ).iterator();
    }

    // send message to every ChatClient
    while ( iterator.hasNext() ) {
        ChatClient client = ( ChatClient ) iterator.next();
        client.deliverMessage( message );
    }

} // end method postMessage

// notify each client that server is shutting down and
// terminate server application
public void stopServer() throws RemoteException
{
    System.out.println( "Terminating server ..." );
    Iterator iterator = null;

} // end method stopServer
```

Fig. 27.7 **ChatServerImpl** implements the Deitel messenger **ChatServer** using RMI-IIOP (part 2 of 3).
// get Iterator for Set of registered clients
synchronized( clients ) {
    iterator = new HashSet( clients ).iterator();
}

// send serverStopping message to every ChatClient
while ( iterator.hasNext() ) {
    ChatClient client = ( ChatClient ) iterator.next();
    client.serverStopping();
    System.err.println( "Disconnected: " + client );
}

// create Thread to terminate application after
// stopServer method returns to caller
Thread terminator = new Thread( new Runnable() {
    // sleep for 5 seconds, print message and terminate
    public void run()
    {
        // sleep
        try {
            Thread.sleep( 5000 );
        }
        // ignore InterruptedExceptions
        catch ( InterruptedException exception ) {}}
        System.err.println( "Server terminated" );
        System.exit( 0 );
    }
});

terminator.start(); // start termination thread

Fig. 27.7 ChatServerImpl implements the Deitel messenger ChatServer using RMI-IIOP (part 3 of 3).

// ChatServerAdministrator.java
// ChatServerAdministrator is a utility for starting and stopping
// the RMI-IIOP ChatServer implementation.
package com.deitel.messenger.rmi_iio.server;

// Java core packages
import java.rmi.*;
import java.rmi.activation.*;

Fig. 27.8 ChatServerAdministrator application for starting and stopping RMI-IIOP ChatServer (part 1 of 4).
import java.util.*;

// Java extension packages
import javax.rmi.*;
import javax.naming.*;

// Deitel packages
import com.deitel.messenger.rmi.server.StoppableChatServer;

public class ChatServerAdministrator {
    
    // set up ChatServer object
    private static void startServer() {
        // register ChatServer in Naming Service
        try {

            // create ChatServerImpl object
            ChatServerImpl chatServerImpl = new ChatServerImpl();

            // create InitialContext for naming service
            Context namingContext = new InitialContext();

            System.err.println( "Binding server to naming service.." );

            // bind ChatServerImpl object to naming service
            namingContext.rebind( "ChatServer", chatServerImpl );

            System.out.println( "Server bound to naming service" );
        } // end try

        // handle exception registering ChatServer
        catch ( NamingException namingException ) {
            namingException.printStackTrace();
            System.exit( 1 );
        }

        // handle exception creating ChatServer
        catch ( RemoteException remoteException ) {
            remoteException.printStackTrace();
            System.exit( 1 );
        }
    } // end method startServer

    // terminate server
    private static void terminateServer() {
        // look up and terminate ChatServer
        try {

        } // end try
    }

    // application for starting and stopping RMI-IIOP ChatServer (part 2 of 4).
// create naming Context for looking up server
Context namingContext = new InitialContext();

// find ChatServer remote object
Object remoteObject =
    namingContext.lookup( "ChatServer" );

// narrow remoteObject to StoppableChatServer
StoppableChatServer chatServer =
    ( StoppableChatServer ) PortableRemoteObject.narrow(
        remoteObject, StoppableChatServer.class );

// stop ChatServer
chatServer.stopServer();

// remove ChatServer from Naming Service
namingContext.unbind( "ChatServer" );

} // end try

// handle exception looking up ChatServer
catch ( NamingException namingException ) {
    namingException.printStackTrace();
}

// handle exception communicating with ChatServer
catch ( RemoteException remoteException ) {
    remoteException.printStackTrace();
}

} // end method terminateServer

// launch ChatServerAdministrator application
public static void main( String args[] )
{
    // check command-line arguments and start or stop server
    if ( args[ 0 ].equals( "start" ) )
        startServer();
    else if ( args[ 0 ].equals( "stop" ) )
        terminateServer();

    // print usage information
    else
        System.err.println(
            "Usage: java ChatServerAdministrator start | stop" );

} // end method main

Fig. 27.8 ChatServerAdministrator application for starting and stopping
RMI-IIOP ChatServer (part 3 of 4).
Figure 27.8 shows text output from ChatServerAdministrator. In this output, ChatServerAdministrator launches the ChatServer and binds it to a naming service. Three clients connect to the naming service—the first client then disconnects. The ChatServerAdministrator terminates the ChatServer, which disconnects the remaining two clients and shuts down the server.
27.7.2 ChatClient RMI-IIOP Implementation

To implement a ChatClient for the RMI-IIOP messenger, we must provide an RMI-IIOP implementation of interface MessageManager. We then must create a new DeitelMessenger application for launching ClientGUI with the new MessageManager implementation.

Class RMIIOOPMessageManager (Fig. 27.9) implements the ChatClient and MessageManager interfaces using RMI-IIOP. Like the RMIMessageManager implementation from Chapter 13, class RMIIOOPMessageManager is a remote object that communicates with the ChatServer remote object. However, class RMIIOOPMessageManager extends class PortableRemoteObject for compatibility with IIOP (line 24). Method connect (lines 38–57) creates an InitialContext for the Naming Service (line 42) and invokes method lookup of interface Context to retrieve a remote reference to the RMI-IIOP ChatServer (lines 45–46). Lines 48–49 invoke method narrow of class PortableRemoteObject to convert the Object remote reference to a ChatServer remote reference. Line 52 invokes method registerClient of interface ChatServer to register the RMIIOOPMessageManager as a ChatServer client. Line 55 sets the MessageListener to which incoming ChatMessages should be delivered. The remainder of class RMIIOOPMessageManager is identical to class RMIMessageManager.

Class DeitelMessenger (Fig. 27.10) launches the Deitel messenger client using classes ClientGUI and RMIIOOPMessageManager. Line 18 creates an instance of class RMIIOOPMessageManager for communicating with the ChatServer. Lines 21–24 create a ClientGUI for the RMIIOOPMessageManager and display the GUI to the user.

```java
// RMIIOOPMessageManager.java
// RMIIOOPMessageManager implements the ChatClient remote
// interface and manages incoming and outgoing chat messages
// using RMI over IIOP.
package com.deitel.messenger.rmi_iiop.client;

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.util.*;

// Java extension packages
import javax.rmi.*;
import javax.naming.*;
import java.rmi.server.*;
import java.util.*;

// RMIIOOPMessageManager implements the ChatClient and
// MessageManager interfaces using RMI-IIOP (part 1 of 3).
```
Fig. 27.9  **RMIIOPMessageManager** implements the **ChatClient** and **MessageManager** interfaces using RMI-IIOP (part 2 of 3).
// notify listener of disconnect
defireServerDisconnected( "" );

} // end method disconnect

// send ChatMessage to ChatServer
public void sendMessage( String fromUser, String message )
  throws Exception
{
  if ( chatServer == null )
    return;

  // create ChatMessage with message text and user name
  ChatMessage chatMessage =
    new ChatMessage( fromUser, message );

  // post message to ChatServer
  chatServer.postMessage( chatMessage );

}  // end method sendMessage

// process delivery of ChatMessage from ChatServer
public void deliverMessage( ChatMessage message )
  throws RemoteException
{
  if ( messageListener != null )
    messageListener.messageReceived( message.getSender(),
    message.getMessage() );
}

// process server shutting down notification
public void serverStopping() throws RemoteException
{
  chatServer = null;
  fireServerDisconnected( "Server shut down." );
}

// register listener for disconnect notifications
public void setDisconnectListener(
    DisconnectListener listener )
{
  disconnectListener = listener;
}

// send disconnect notification
private void fireServerDisconnected( String message )
{
  if ( disconnectListener != null )
    disconnectListener.serverDisconnected( message );
}
27.7.3 Compiling and Running the ChatServer and ChatClient

Compile ChatServerImpl, ChatServerAdministrator, RMIIOPMessageManager and DeitelMessenger using the Java compiler. You must also compile the RMI-IIOP stub classes for the ChatServerImpl and RMIIOPMessageManager remote objects. Use the rmic tool with the -iiop command-line option to generate the appropriate stubs, for example,

```
rmic -iiop com.deitel.messenger.rmi_iiopt.server.ChatServerImpl
```

Running the RMI-IIOP Deitel messenger system requires a CORBA Naming Service.

For this example, we use tnameserv, which we used for the examples in Chapter 26. Start tnameserv by typing the following at a command prompt:

```
tnameserv -ORBInitialPort 1050
```

Class ChatServerAdministrator launches the ChatServer and registers the server with the Naming Service. To start ChatServer, type the following at a command prompt:

```
ChatServerAdministrator
```
java -Djava.naming.factory.initial=com.sun.jndi.cos-naming.CNCtxFactory
   -Djava.naming.provider.url=iiop://localhost:1050
   com.deitel.messenger.rmi_iiop.server.ChatServerAdministrator start

The `-D` command-line option specifies system properties for the virtual machine. Recall that property `java.naming.factory.initial` specifies the class that implements the Naming Service’s `InitialContext`. Property `java.naming.provider.url` specifies the location and port where the Naming Service is running. For this example, the Naming Service runs on the local machine on port **1050**.

Once **ChatServer** is running and bound to the Naming Service, launch several clients by typing the following at a command prompt and executing repeatedly:

```
java -Djava.naming.factory.initial=com.sun.jndi.cos-naming.CNCtxFactory
   -Djava.naming.provider.url=iiop://localhost:1050
   com.deitel.messenger.rmi_iiop.client.DeitelMessenger
```

### 27.8 Future Directions

The Object Management Architecture defined the various levels of object services in the first version of a full CORBA architecture back in 1990. The OMA is the guiding document for the design of many CORBA systems, yet falls short in average-size systems that still have needs for distributed system concepts. CORBA and OMA expect developers to use the reference architecture’s flexibility to make the same fine-grained decisions for each system being built. (How should the POAs be configured? What activation policies should be enabled and when? Is there a process for choosing the proper CORBA services and integrating them into a general, reusable framework?)

The CORBA Component Model and Enterprise JavaBeans abstract many system issues through programming constructs such as containers and subsystems. CCM guarantees Enterprise JavaBeans’ place in the future by allowing future programming languages to interoperate with Java transparently, so Java can interoperate with COBOL, C, C++ and other IDL-mapped languages.

OMG is now working on an architectural process called the Model Driven Architecture™ (MDA). In late 2000 and early 2001, OMG members noticed that the occasional introduction of new middleware platforms turned into a steady stream which was not likely to slow or stop anytime in the foreseeable future. To restore stability to standards and the enterprise development process, OMG created and adopted the MDA as the basis for its future specifications. In the MDA, a specification or application is defined first and fundamentally as a Platform-Independent Model (PIM) in the Unified Modeling Language (UML). MDA tools generate one or more Platform-Specific Models (PSM) from the application’s PIM, following OMG-standardized mappings. In a third step, the MDA tool generates interfaces, stub and template code, configuration files, build and deployment files and application code from the PSM. MDA supports, and OMG will publish standards in, many middleware platforms such as EJB, XML/SOAP, C#.NET and CORBA. Interoperability is provided by bridges or by bridging code generated by MDA tools from models. The PIM retains its value as the technology around it shifts over time. Industry standards groups, including especially OMG’s Domain Task Forces, shifted quickly to MDA for their
work because specifications formulated as PIMs embody only business functionality and behavior, and so outlast individual technologies and trends.

This expansion of OMG’s architecture beyond CORBA (which receives the same level of support as before MDA) is giving new vitality to OMG and its commitment to technology-independent domain standards.

### 27.9 Internet and World Wide Web Resources

**www.omg.org/cgi-bin/doc?formal/99-07-59**
The OMG Java-to-IDL Specification describes in detail the various mappings and limitations inherent in using this reverse IDL mapping. This document is also available in Postscript and PDF.

**www.omg.org/technology/documents/formal/corba_services_available_electro.htm**
This OMG page lists the CORBA services specifications that are available electronically. Downloads of each specification are available as Postscript and PDF files.

**www.omg.org/cgi-bin/doc?orbos/98-01-18**
Objects-by-Value, as a distinguishing characteristic between RMI and CORBA, is explained in this document. The specification describes applicable keywords, mechanism concepts and how OBV applies to Java.

**www.w3.org/TR/NOTE-ice**
This site gives the definitive description and thinking of Open Software Description Format. An understanding of XML and Web technologies is necessary to appreciate what OSD (and by extension the Information and Content Exchange Protocol) is trying to do.

**openorb.exolab.org/openorb.html**
In the spirit of open source, ExoLab Group has released (and updates) a full, open-source implementation of CORBA 2.4.1 for the development of distributed systems. OpenORB supports both POAs and BOAs and includes various tools (such as IDL compilers).

**openorb.exolab.org/services.html**
Although not a full set of CORBA services, this open-source alternative offers a collection of useful services for developers building CORBA frameworks. Naming, Persistence and Transaction are available (to name a few). Follow the Extensions link to find their implementation of RMI-IIOP.

**www.omg.org/cgi-bin/doc?formal/01-06-07**
This Web page provides a link to the June 2001 paper, *Java Language to IDL Mapping*.

**www.omg.org/cgi-bin/doc?formal/98-12-09**

**www.omg.org/cgi-bin/doc?ptc/99-10-03**
This page on the Object Management Group’s Web site offers the Component FTF Edited Drafts of CORBA Core Chapters, originally published October 1999.

**www.omg.org/technology/documents/formal/internationalization_and_time.htm,**
The January 2000 paper, *Internationalization, Time Operations, and Related Facilities*, is available on this page.

**www.omg.org/technology/documents/formal/mobile_agent_facility.htm**
The Mobile Agent Facility Specification is available at this link.

**www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf**
This site contains an excellent collaborative work, *Overview of the CORBA Component Model*. 
Learn-by-doing is the approach taken by this well-written tutorial explaining how to use RMI-IIOP. The RMI-IIOP Programmer’s Guide includes many links for background information useful to developers new to RMI, CORBA and IIOP.

**CORBA Component Model Resources**

corbaweb.lifl.fr/OpenCCM/
The OpenCCM Platform is an open-source implementation of the CORBA Component Model. Sponsored by the Laboratoire de Recherche en Informatique de l’Université des Sciences et Technologies de Lille (LIFL) in France, this open-source implementation of the CORBA Component Model is a worthwhile place for CORBA developers to inexpensively try out CCM technology.

www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf
This document is an overview of the CORBA Component Model and why it is a worthwhile addition to the CORBA marketplace. This well-written overview goes into more detail about containers, components, deployment and the runtime environment a developer can expect from CCM.

**Model Driven Architecture™ Resources**

www.omg.org/mda/index.htm
Model Driven Architecture promises to leverage many existing technologies into a cohesive view allowing the development of large-grained systems. The documents at this link explain the direction in which the OMG expects MDA to go.

**TERMINOLOGY**

- **abstract**
- ACID
- activation
- Asynchronous Method Invocation (AMI)
- Basic Object Adapter (BOA)
- call-by-value
- catalog
- Component Implementation Framework (CIF)
- Component Interface Definition Language (CIDL)
- ComponentHome
- component-managed persistence
- ConsumerAdmin
- Container programming model
- container-managed persistence
- conversational container
- CORBA Component Model (CCM)
- CORBA services
- durable container
- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI)
- emits
- Enterprise JavaBeans
- **entity** component
- EventChannel
- EventChannelFactory
- Event Service
- flat transactions
- HomeFinder
- Interface Repository (IR)
- Java Naming and Directory Interface (JNDI)
- Model Driven Architecture™
- NameComponent
- Naming
- NamingContext
- nested transactions
- Notification Service
- Object activation
- Object Constraint Language (OCL)
- Object Transaction Service (OTS)
- Object-by-Reference
- Object-by-Value
- Open Software Description Format (OSD)
- OperationDef
- passivation
- Persistent State Definition Language (PSDL)
- Persistent State Service (PSS)
- Portable Object Adapter (POA)
SELF-REVIEW EXERCISES

27.1 Fill in the blanks in each of the following statements:
   a) An association between a name and a value is called a ________.
   b) In the context of transaction processing, ACID stands for ________, ________, ________, and ________.
   c) Use of the Event Service is done through the ________.
   d) In RMI-IIOP, the ________ is used in place of the java.rmi.UnicastRemoteObject.
   e) The DII allows ________-typed calls to be made from a client.
   f) ________ give components an environment to exist in and available services to use.

27.2 State whether each of the following is true or false. If false, explain why.
   a) RMI uses the same underlying communication protocol as CORBA.
   b) TIEs are POAs that use the delegation model.
   c) The Event Service allows only one delivery model at a time: Either pull or push, but not both.
   d) Session CORBA components are equivalent to EJB stateless session beans.
   e) Session CORBA components are equivalent to EJB stateful session beans.
   f) Enterprise JavaBeans specify the use of an Event Service.
   g) CCM supports multithreaded containers.

ANSWERS TO SELF-REVIEW EXERCISES

27.1 a) name binding. b) Atomic, Consistent, Isolated, Durable. c) EventChannelFactory. d) javax.rmi.PortableRemoteObject. e) dynamically. f) Containers.

27.2 a) False. RMI uses the Java Remote Method Protocol (JRMP), CORBA uses the Internet Inter-ORB Protocol (IIOP). b) True. c) False. The Event Service allows a mix of either pull or push in any combination by the suppliers and consumers. d) True. e) True. f) False. The EJB specification requires only Naming, Security, Persistence and Transaction. g) True.
EXERCISES

27.3 Write a `QuoteService` that works in either a push or pull model: define an operation to allow clients to register and receive `Quotes` (push) and define another operation to return a `Quote` for a specific `Symbol` (pull). Write a test client that gets pushed by the server and also pulls `Quotes` from the server.

27.4 Write a GUI client that displays the `Quote` information in a `JTable`. Use a `TableModel` to mediate access to the `Quote` data. Use a client-side proxy to wrap the remote `QuoteService`. Use a client-side proxy to wrap the `Quote` object (if necessary).

27.5 Modify the `QuoteService` to return an array of `Quotes` based on an array of incoming `Symbol` names.

27.6 Write a `LoggingService`. One model for logging messages is to connect to an object that would deliver messages in an appropriate fashion based on how important they are. The API should take into account the source, priority, topic and contents. System architecture would consider priority a quality-of-service attribute. The `LoggingService` should be a write-only service (messages go in, but not out). Take multiuser access into account in the API. What kind of activation policies can a POA be assigned to handle some of these issues?

27.7 From an implementation perspective, the `LoggingService` could distribute the delivery of the message in a number of ways. This is a classic trade-off between making the API straightforward (connect to the service and post messages directly to it) or making its use more complex, but more flexible (make the service more of a pool of connections usable by many clients). Change the name of the `LoggingService` to `LoggingConnection` and have it return a `LogStream` object. The `LogStream` takes the source, priority, topic and contents of the message into account (just as the previous example did), only now the load balancing is more explicit.

WORKS CITED

BIBLIOGRAPHY

Object Management Group Resources


Objectives

- To understand peer-to-peer application architectures.
- To understand how various popular peer-to-peer applications work.
- To create a complete P2P instant-messenger application using RMI and Jini technologies.
- To create a complete P2P instant-messenger application using Multicast Sockets and RMI.
- To introduce the emerging JXTA peer-to-peer technology.

No! let me taste the whole of it, fare like my peers,
The heroes of old,
Bear the brunt, in a minute pay glad life’s arrears
Of pain, darkness, and cold.
Robert Browning

If we do not lay out ourselves in the service of mankind whom should we serve?
John Adams

We are as much as we see...
Henry David Thoreau

Good counselors lack no clients.
William Shakespeare
28.1 Introduction

Instant-messaging systems and document-sharing applications such as AOL Instant Messenger™ and Gnutella have exploded in popularity, transforming the way users interact with one another over networks. In a peer-to-peer (P2P) application, each node performs both client and server functions. Such applications distribute processing responsibilities and information to many computers, thus reclaiming otherwise wasted computing power and storage space, and eliminating central points of failure.

In this chapter, we introduce the fundamental concepts of peer-to-peer applications. Using Jini (Chapter 22), RMI (Chapter 13) and multicast sockets, we present two peer-to-peer application case studies. We first implement an instant-messaging application with Jini and RMI to demonstrate a more substantial Jini application and show the benefits of integrating Jini with other technologies. We then implement the same instant-messaging application using multicast sockets and RMI. Finally, we introduce JXTA (short for “juxtapose”)—a new open-source technology from Sun Microsystems™ that defines common protocols for implementing peer-to-peer applications.

28.2 Client/Server and Peer-to-Peer Applications

Many network applications operate on the principle that computers should be segregated by function. Some computers—called servers—offer common stores of programs and data. Other computers—called clients—access the data provided by the servers. The Yahoo!™ search engine (www.yahoo.com) is an example of a client/server application. Client send queries to the central servers, which have pre-compiled catalogs of the Internet. The central servers refer to their databases and respond with the requested information.
P2P applications are different than client/server applications. Instead of segregating computers by function, all computers act as both clients and servers. P2P applications are similar to the telephone system—a single user can both speak (send information) and listen (receive information). Figure 28.1 lists some common peer-to-peer applications.

### 28.3 Centralized vs. Decentralized Network Applications

An application that uses a *centralized server* exemplifies the client/server relationship. One major weakness of this centralized system is the dependency on the central server. If the central node (i.e., server) fails, the entire application also fails. The server’s capabilities limit the application’s overall performance. For instance, Web sites can fail when malicious users overload the Web server(s) with an excessive number of requests. However, centralized architectures also have advantages, such as simplifying management tasks (e.g., monitoring user access by providing single points of network control).

True P2P applications are completely decentralized and do not suffer from the same deficiencies as applications that depend on centralized servers. If nodes in a P2P application fail, well-designed P2P applications continue to function. P2P applications often leverage distributed computational power. *Freenet*, for instance, allows users to share documents in a way that precludes censorship. Peer-to-peer architectures allow real-time searches that return up-to-date results. Centralized search engines today are slow to incorporate recently created Web data into these catalogs. Peer-to-peer searches accurately reflect the network status at the time of the query.

<table>
<thead>
<tr>
<th>Distributed Applications</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gnutella</td>
<td>A P2P technology that does not use any central servers. There is no authentication, and peers search for files via a distributed-search mechanism.</td>
</tr>
<tr>
<td>KaZaA</td>
<td>A file-sharing application that is a hybrid between Gnutella and centralized applications. A central server authenticates all users. Certain peers serve as search hubs, which catalog the files of peers connected to them. Searches are distributed to each search hub, which then respond with results that allow direct connections for file transfers.</td>
</tr>
<tr>
<td>Instant Messengers</td>
<td>Peer-to-peer applications that enable users to send short text messages and files to one another. Most instant messengers use central servers that authenticate all users and route messages between peers.</td>
</tr>
<tr>
<td>Telephone System</td>
<td>A peer-to-peer application that enables users to conduct voice conversations remotely.</td>
</tr>
</tbody>
</table>

Fig. 28.1 Common P2P applications.

Peer-to-peer applications have disadvantages as well. Anyone with the appropriate software can join the network of peers and often remain anonymous—for this reason, determining who is on the network at any instant is difficult. Also, the lack of a central server hinders the enforcement of copyright and intellectual-property laws. Real-time searches can be slow and increase network traffic, because every query must propagate throughout the entire network.

True client/server applications are completely centralized, whereas true peer-to-peer applications are completely decentralized. Many applications adopt aspects of both to achieve specific goals. For example, some file-sharing applications are not true peer-to-peer applications, because they use central servers to authenticate users and index each peer’s shared files. However, peers connect directly to one another to transfer files. In such a system, centralization increases search performance but makes the network dependent on a central server. Performing file transfers between peers decreases the load on the central server.

### 28.4 Peer Discovery and Searching

Peer discovery is the act of finding peers in a P2P application. Decentralizing an application often slows peer discovery and information searching. Gnutella presents one approach for circumventing these problems. Gnutella is a true peer-to-peer technology that enables distributed information storage and retrieval. Users can search for and download files from any peer on the network. Users first join a Gnutella network by specifying the network address of a known Gnutella peer. Without knowing of at least one peer on the network, a user cannot join the network. Each user’s Gnutella software functions as a server and uses the HTTP protocol to search for and transfer files.

To perform a search, a peer sends search criteria to several nearby peers. Those peers then propagate the search throughout the network of peers. If a particular peer can satisfy the search, that peer passes this information back to the originator. The originator then connects directly to the target peer and downloads the information. The peer that made the original query loses anonymity only when it connects directly to the peer with the requested file to begin file transfer.

In the Freenet P2P application, files also propagate throughout the network of peers. Each peer that uses Freenet forwards search requests to only one other peer. If the search fails, the peer that received the request forwards the request to the next known peer. If we view the searching peer as the root of a hierarchical structure that represents the Gnutella network, each search request traverses in a breadth-first fashion, because peers forward each search request to several peers at once. Freenet works essentially like Gnutella, except that each search request traverses depth-first.

Searches conducted in both Gnutella and Freenet are called distributed searches. Distributed searches make networks more robust by removing single points of failure, such as central servers. Information found via distributed searches is current because it reflects the current state of the network. Not only can peers find information in this way, but peers can search for other peers via distributed searches as well.

### 28.5 Case Study: Deitel Instant Messenger

In the next several sections, we present a peer-to-peer application that allows users to send instant messages to one other. The Deitel Instant Messenger application uses Jini to boot-
strap—or register—users onto the peer-to-peer network. Jini lookup services store remote references to peers on the network. Peers use RMI to connect to one another directly and converse. Although we sometimes refer to an instance of the application as a client, each application instance is both a client and a server.

The application’s main window (Fig. 28.2, left-side image) displays a list of peers running Deitel Instant Messenger on the local network. To send an instant message, the user selects a name and clicks the **Connect** button. The conversation window (Fig. 28.2, right-side image) appears with the selected peer’s name as the title. The user can type a message, then send it by clicking **Send**.

Deitel Instant Messenger uses Sun’s Jini technology, which requires at least one lookup service. However, with only one lookup service, the application acts as a hybrid between P2P and client/server applications. The lookup service is centralized to enable peers to find other peers easily. Peers use RMI to connect to one another directly. A setup with only one lookup service is similar to the setup that many instant messaging applications use today.

For Deitel Instant Messenger to be truly peer-to-peer, each node must run its own lookup service. However, running a lookup service on each peer is inefficient and could generate extensive network traffic. Thus, a compromise between using only one lookup service and using a lookup service for every node balances both reliability and robustness with speed and efficiency.

If a node does not run a lookup service, that node depends on the existing lookup services on the network—this results in centralization. Each client consists of a Jini-service proxy and an RMI object that enables the peer-to-peer communication. The application registers with all known lookup services, which includes those found through both multicast and unicast discovery. Users can add lookup services to the program by selecting the **Add Locator** item in the **File** menu and supplying the lookup service’s URL.

After registering the client’s proxy with the lookup service, the Deitel Instant Messenger client retrieves all other proxies in the Jini lookup service. These proxies represent all known peers. To send messages, a client must hold a remote reference to the other peer. Therefore, to begin a conversation, a client sends a reference to itself through the service proxy to the other peer. The other peer responds by sending a remote reference back to the peer that initiated the conversation. When each party has a reference to the other, they can both send and receive messages.

---

**Fig. 28.2** Sample windows of Deitel Instant Messenger.
The major steps in this example are as follows:

1. define a service interface that contains a remote reference to the service implementation,
2. define the service implementation,
3. provide methods for bootstrapping the service into the peer group and
4. compile and run the P2P application.

We discuss each step in detail in the next several sections, as we implement the Deitel Instant Messenger application.

### 28.6 Defining the Service Interface

The first step in the example is to define the service interface—IMService (Fig. 28.3). Method `connect` (lines 16–17) enables remote users to send a remote reference to an IMPeer (the instant-messenger peer). A remote reference enables one-way communication. To establish two-way communication, each client must have a remote reference to the other client’s IMPeer objects.

IMPeer (Fig. 28.4) specifies the interface for communicating between peers. Method `connect` (line 16) takes as an argument an IMPeer and returns an IMPeer reference. Remote interface IMPeer describes the basic methods for interacting with an IMPeer.

```java
// IMService.java
// IMService interface defines the methods
// through which the service proxy
// communicates with the service.
package com.deitel.advjhtpl.jini.IM.service;

// Java core packages
import java.rmi.*;

// Deitel packages
import com.deitel.advjhtpl.jini.IM.IMPeer;

public interface IMService extends Remote {
    // return RMI reference to a remote IMPeer
    public IMPeer connect( IMPeer sender )
        throws RemoteException;
}
```

Fig. 28.3 Interface IMService specifies how service proxy interacts with the service.

```java
// IMPeer.java
// Interface that all Peer to Peer apps must implement
package com.deitel.advjhtpl.jini.IM;
```

Fig. 28.4 Interface IMPeer specifies interaction between peers (part 1 of 2).
In line 9, `IMPeer` extends interface `java.rmi.Remote`, because `IMPeer`s are remote objects. The Deitel Instant Messenger client sends a message to a peer by calling that peer’s `sendMessage` method (line 12–13) and passing a `Message` object as an argument. Class `Message` (Fig. 28.5) represents a message that `IMPeer`s can send to one another.

```java
// Message.java
// Message represents an object that can be sent to an IMPeer;
// contains the sender and content of the message.
package com.deitel.advjhtp1.jini.IM;

// Java core package
import java.io.Serializable;

public class Message implements Serializable {

    private static final long SerialVersionUID = 20010808L;
    private String from;
    private String content;

    // Message constructor
    public Message( String messageSenderName, String messageContent )
    {
        from = messageSenderName;
        content = messageContent;
    }

    // get String representation
    public String toString()
    {
        return from + "":" + content + "\n";
    }
}
```

**Fig. 28.4** Interface `IMPeer` specifies interaction between peers (part 2 of 2).

**Fig. 28.5** Class `Message` defines an object for sending and receiving messages between peers (part 1 of 2).
In the upcoming exercises, we ask you to extend this class to allow more complex types of communication. Line 10 specifies that class Message implements interface Serializable, because Messages must be serialized for delivery over RMI. The Message constructor (lines 16–21) takes as arguments the sender’s name and the content of the message.

### 28.7 Defining the Service implementation

The second step in the example is defining the service implementation—IMServiceImpl (Fig. 28.6), which implements interface IMServe. Lines 18–19 declare that class IMServiceImpl extends UnicastRemoteObject, which facilitates exporting the IMServiceImpl as a remote object.
The second constructor (lines 28–31) takes as a String argument the user’s name. This name appears in the PeerList window. Lines 40–56 implement method connect.

For two peers to communicate, each peer requires a remote reference to the other peer. The following steps summarize the connection process:

1. Peer A sends a reference to itself to peer B by invoking IMService method connect.
2. Peer B stores that reference (line 51) to peer A for use when conversation starts.
3. Peer B returns a reference to itself (line 54) to peer A.

The Deitel Instant Messenger creates an object of class IMPeerListener (Fig. 28.7) — the GUI that starts the peer communication. The upper text area outputs messages sent via the remote reference to an IMPeer. The bottom text area contains text to be sent via a remote method call.

```java
private static final long serialVersionUID = 20010808L;
private String userName = "Anonymous";

// IMService no-argument constructor
public IMSServiceImpl() throws RemoteException{}

// IMService constructor takes userName
public IMSServiceImpl( String name ) throws RemoteException
{
    userName = name;
}

// sets serviceís userName
public void setUserName( String name )
{
    userName = name;
}

// return RMI reference to an IMPeer on the receiver side
public IMPeer connect( IMPeer sender )
throws RemoteException
{
    // Make a GUI and IMPeerImpl to be sent to remote peer
    IMPeerListener listener =
        new IMPeerListener( userName );

    IMPeerImpl me = new IMPeerImpl( userName );
    me.addListener( listener );

    // add remote peer to my GUI
    listener.addPeer( sender );

    // send my IMPeerImpl to him
    return me;
}
```

Fig. 28.6 IMSServiceImpl service implementation for our case study (part 2 of 2).
Fig. 28.7 Class **IMPeerListener** is the GUI that starts peer communication (part 1 of 4).
JPanel panel = new JPanel();
panel.setLayout( new BorderLayout( 5, 5 ) );
panel.add( new JScrollPane( messageArea ),
BorderLayout.CENTER );

// create JTextArea for entering new messages
inputArea = new JTextArea( 4, 12 );
inputArea.setLineWrap( true );
inputArea.setWrapStyleWord( true );

// map Enter key in inputArea area to sendAction
Keymap keyMap = inputArea.getKeymap();
KeyStroke enterKey = KeyStroke.getKeyStroke(
    KeyEvent.VK_ENTER, 0 );
keyMap.addActionForKeyStroke( enterKey, sendAction );

// lay out inputArea and sendAction JButton in Box
Box box = Box.createVerticalBox();
box.add( new JScrollPane( inputArea ) );
box.add( new JButton( sendAction ) );
panel.add( box, BorderLayout.SOUTH );

// lay out components
Container container = getContentPane();
container.add( panel, BorderLayout.CENTER );
setSize( 200, 400 );
setVisible( true );
}

// Action for sending messages
private class SendAction extends AbstractAction {

    // configure SendAction
    public SendAction()
    {
        putValue( Action.NAME, "Send" );
        putValue( Action.SHORT_DESCRIPTION,
            "Send Message" );
        putValue( Action.LONG_DESCRIPTION,
            "Send an Instant Message" );
    }

    // send message and clear inputArea
    public void actionPerformed( ActionEvent event )
    {
        // send message to server
        try {
            Message message = new Message( userName,
                inputArea.getText() );
        }
    }
}

Fig. 28.7 Class IMPeerListener is the GUI that starts peer communication (part 2 of 4).
```java
// use RMI reference to send a Message
remotePeer.sendMessage( message );

// clear inputArea
inputArea.setText( "" );
displayMessage( message );

// catch error sending message
try ( RemoteException remoteException ) {
    JOptionPane.showMessageDialog( null, "Unable to send message." );
    remoteException.printStackTrace();
} // end method actionPerformed

public void displayMessage( Message message ) {
    // displayMessage uses SwingUtilities.invokeLater
    // to ensure thread-safe access to messageArea
    SwingUtilities.invokeLater( new MessageDisplayer( message.getSenderName(), message.getContent() ) );
}

private class MessageDisplayer implements Runnable {
    private String fromUser;
    private String messageBody;

    // MessageDisplayer constructor
    public MessageDisplayer( String from, String body ) {
        fromUser = from;
        messageBody = body;
    }

    // display new message in messageArea
    public void run() {
        // append new message
        messageArea.append( "\n" + fromUser + " > " + messageBody );
    }
}
```

**Fig. 28.7** Class IMPeerListener is the GUI that starts peer communication (part 3 of 4).
In Fig. 28.6, line 48 adds an `IMPeerListener` object to an `IMPeerImpl` object using method `addListener`. `IMPeerImpl` method `sendMessage` sends `Message` objects to the `IMPeerListener`.

Line 51 calls method `addPeer` to add a reference to a remote `IMPeer` object to the `IMPeerListener`. This allows the `IMPeerListener` to send messages to the remote peer. Notice the symmetry: `IMPeerListener` is also client/server, because the application itself is client/server. Just as each peer must provide a reference to itself and store references to the remote party, so must the `IMPeerListener`. `IMPeerListener` mediates between the two peers.

`IMPeerListener` (Fig. 28.7) is the GUI for communication between peers. Method `addPeer` (lines 167–173) stores a reference to the remote `IMPeer` and sets the title of the conversation window to the name of the remote `IMPeer`. When the user presses the `JButton` on the GUI, line 107 calls method `sendMessage` of the remote `IMPeer`, passing the contents of `inputArea` as the argument. This is how the client sends messages to remote peers. On the receiving end, the remote `IMPeerImpl` (Fig. 28.8) calls `IMPeerListener` method `displayMethod` to display the message.
Class `IMPeerImpl` implements the `IMPeer` interface. An `IMPeerImpl` instance represents each peer in conversations with other peers and allows peers to communicate with one another. Lines 17–18 declare that `IMPeerImpl` extends `UnicastRemoteObject`, to facilitate exporting the `IMPeerImpl` as a remote object. Method `addListener` (lines 35–38) adds an object of type `IMPeerListener` that will display the `IMPeerImpl`’s actions. Method `sendMessage` (lines 41–45) calls method `displayMessage` of interface `IMPeerListener` to display the message.
28.8 Registering the Service

The third step in the example is registering (bootstrapping) the service with the peer group. Class `IMServiceManager` (Fig. 28.9) takes the user’s name as a `String` argument and uses the Jini’s `JoinManager` class to register the service with all known lookup services. The code is similar to the `JoinManager` version of the `SeminarInfo` program in Chapter 22. One difference is that the constructor (lines 33–65) takes a `String` that specifies the `Name Entry` for the service.

```java
// IMServiceManager.java
// IMServiceManager uses JoinManager to find Lookup services,
// registers the IMService with the Lookup services,
// manages lease renewal
package com.deitel.advjhtp1.jini.IM;

// Java core packages
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.io.IOException;

// Jini core packages
import net.jini.core.lookup.ServiceID;
import net.jini.core.entry.Entry;

// Jini extension packages
import net.jini.lookup.entry.Name;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.JoinManager;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceIDListener;

// Deitel packages
import com.deitel.advjhtp1.jini.IM.service.;

public class IMServiceManager implements ServiceIDListener {

    JoinManager manager;
    LookupDiscoveryManager lookupManager;
    String serviceName;

    // constructor takes name of the service
    public IMServiceManager( String screenName ) {
        System.setSecurityManager( new RMISecurityManager() );
        // sets the serviceName of this service
        serviceName = screenName;
    }

    // // IMServiceManager (part 1 of 3).
```
// use JoinManager to register SeminarInfo service
// and manage lease
try {

    // create LookupDiscoveryManager for discovering
    // lookup services
    lookupManager =
        new LookupDiscoveryManager( new String[] { "" },
        null, null );

    // create and set Entry name for service
    // name used from constructor
    Entry[] entries = new Entry[ 1 ];
    entries[ 0 ] = new Name( serviceName );

    // create JoinManager
    manager = new JoinManager( createProxy(),
        entries, this, lookupManager,
        new LeaseRenewalManager() );
}

    // handle exception creating JoinManager
    catch ( IOException exception ) {
        exception.printStackTrace();
    }
}

    // return the LookupDiscoveryManager created by JoinManager
public LookupDiscoveryManager getDiscoveryManager()
{
    return lookupManager;
}

    // create service proxy
private IMService createProxy()
{
    // get SeminarProxy for SeminarInfo service
    try {
        return( new IMServiceImpl( serviceName ) );
    }

    // handle exception creating SeminarProxy
    catch ( RemoteException exception ) {
        exception.printStackTrace();
    }

    return null;
}

} // end method createProxy

Fig. 28.9 Class IMServiceManager registers IMServiceImpl with lookup services (part 2 of 3).
28.9 Find Other Peers

Class PeerList (Fig. 28.10)—the main window of the Deitel Instant Messenger—lists peers to whom the user can send instant messages. Line 264 creates a new IMServiceManager, passing String userName as an argument to the IMServiceManager constructor to name the peer.

IMPeer (Fig. 28.4) specifies the interface for communicating between peers. Method connect (line 16) takes as an argument an IMPeer and returns an IMPeer reference. Remote interface IMPeer describes the basic methods for interacting with an IMPeer.

A LookupCache also allows an application to respond to services being added, removed, or changed without constantly polling lookup services. The Peer List uses a LookupCache to determine when other peers join and leave the network. To use this functionality, an application must pass a ServiceDiscoveryListener object to method createLookupCache. PeerList implements ServiceDiscoveryListener (lines 33–34), so line 276 passes the this reference to method createLookupCache. ServiceDiscoveryManager uses remote events registered with lookup services to implement asynchronous notification of ServiceDiscoveryEvents. Therefore, we must provide lookup services with a way to download the stub class files for the RemoteEvent handlers. Section 28.10 explains how to do this.

```java
public void serviceIDNotify( ServiceID serviceID )
{
    System.err.println( "Service ID: " + serviceID );
}

public void logout()
{
    manager.terminate();
}
```

Fig. 28.10 Class PeerList is the GUI for finding peers (part 1 of 7).
```java
// Java extension packages
import javax.swing.*;
import javax.swing.event.*;

// Jini core packages
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.*;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;
import net.jini.core.discovery.LookupLocator;

// Deitel Packages
import com.deitel.advjhtp1.jini.IM.service.IMService;
import com.deitel.advjhtp1.jini.IM.client.IMPeerListener;

public class PeerList extends JFrame implements ServiceDiscoveryListener {

    private DefaultListModel peers;
    private JList peerList;
    private List serviceItems;
    private ServiceDiscoveryManager serviceDiscoveryManager;
    private LookupCache cache;
    private IMServiceManager myManager;
    private LookupDiscoveryManager lookupDiscoveryManager;

    // initialize userName to anonymous
    private String userName = "anonymous";

    // method called when ServiceDiscoveryManager finds
    // IM service adds service proxy to serviceItems
    // adds Service name to ListModel for JList
    public void serviceAdded( ServiceDiscoveryEvent event )
    {
        // get added serviceItem
        ServiceItem item = event.getPostEventServiceItem();
        Entry attributes[] = item.attributeSets;

        // iterates through attributes to find name
        for ( int i = 0; i < attributes.length; i++ )
        {
            if ( attributes[ i ] instanceof Name ) {
                System.out.println( "Added: " + item );
                serviceItems.add( item.service );
                peers.addElement( ((Name)attributes[ i ]).name );
                break;
            }
        }
    } // end method serviceAdded
```

Fig. 28.10 Class PeerList is the GUI for finding peers (part 2 of 7).
// empty method ignores serviceChanged event
public void serviceChanged( ServiceDiscoveryEvent event )
{
}

// removes services from PeerList GUI and data structure
// when serviceRemoved event occurs
public void serviceRemoved( ServiceDiscoveryEvent event )
{
    // getPreEvent because item has been removed
    // getPostEvent would return null
    ServiceItem item = event.getPreEventServiceItem();
    Entry attributes[ ] = item.attributeSets;

    // debug
    System.out.println( "Remove Event!" );

    // remove from arraylist and DefaultListModel
    int index = serviceItems.indexOf( item.service );

    // print name of person removed
    if ( index >= 0 )
    {
        System.out.println( "Removing from List:" +
                          serviceItems.remove( index ));

        System.out.println( "Removing from DefList" +
                          peers.elementAt( index ) );

        peers.removeElementAt( index );
    }
} // end method ServiceRemoved

// constructor
public PeerList()
{
    super( "Peer List" );

    System.setSecurityManager( new RMISecurityManager() );

    // get desired userName
    userName = JOptionPane.showInputDialog( PeerList.this, "Please enter your name: " );

    // change title of window
    setTitle( userName + "'s Peer List Window" );

    // Init Lists
    serviceItems = new ArrayList();
    peers = new DefaultListModel();
```java
// init components
peerList = new JList( peers );
peerList.setVisibleRowCount( 5 );
JButton connectButton = new JButton( "Connect" );

// do not allow multiple selections
peerList.setSelectionMode(
    ListSelectionModel.SINGLE_SELECTION );

// set up event handler for connectButton
connectButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            int itemIndex = peerList.getSelectedIndex();
            Object selectedService =
                serviceItems.get( itemIndex );
            IMService peerProxy =
                ( IMService )selectedService;

            // send info to remote peer
            // get RMI reference
            try {
                // set up gui and my peerImpl
                IMPeerListener gui =
                    new IMPeerListener( userName );
                IMPeerImpl me =
                    new IMPeerImpl( userName );
                me.addListener( gui );

                // Connect myGui to remote IMPeer object
                IMPeer myPeer = peerProxy.connect( me );
                gui.addPeer( myPeer );
            }
            catch( RemoteException re ) {
                JOptionPane.showMessageDialog
                    ( null, "Couldn't Connect" );
                re.printStackTrace();
            }
        }
    }); // end connectButton actionListener

// set up File menu
JMenu fileMenu = new JMenu( "File" );
fileMenu.setMnemonic( 'F' );
```

Fig. 28.10 Class PeerList is the GUI for finding peers (part 4 of 7).
// about Item
JMenuItem aboutItem = new JMenuItem("About...");
aboutItem.setMnemonic('A');
aboutItem.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            JOptionPane.showMessageDialog(PeerList.this,
                "Deitel Instant Messenger",
                "About", JOptionPane.PLAIN_MESSAGE);
        }
    });

fileMenu.add(aboutItem);

// AddLocator item
JMenuItem federateItem =
    new JMenuItem("Add Locators");
federateItem.setMnemonic('L');
federateItem.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            // get LookupService url to be added
            String locator =
                JOptionPane.showInputDialog(PeerList.this,
                    "Please enter locator in this" +
                    "form: jini://host:port/");

            try {
                LookupLocator newLocator =
                    new LookupLocator(locator);

                // make one element LookupLocator array
                LookupLocator[] locators = { newLocator };

                // because addLocators takes array
                lookupDiscoveryManager.addLocators(locators);
            }
            catch( MalformedURLException urlException) {
                JOptionPane.showMessageDialog(PeerList.this,
                    "invalid url");
            }
        }
    });

fileMenu.add(federateItem);

Fig. 28.10 Class PeerList is the GUI for finding peers (part 5 of 7).
// set up JMenuBar and attach File menu
JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);
setJMenuBar(menuBar);

// handle window closing event
addWindowListener(
   new WindowAdapter(){
      public void windowClosing(WindowEvent w){
         System.out.println( "CLOSING WINDOW" );
         // disconnects from lookup services
         myManager.logout();
         System.exit(0);
      }
   }
);

// lay out GUI components
peerList.setFixedCellWidth(100);
JPanel inputPanel = new JPanel();
inputPanel.add(connectButton);

container.add(new JScrollPane(peerList), BorderLayout.NORTH);
container.add(inputPanel, BorderLayout.SOUTH);

setSize(100, 170);
setVisible(true);

// peer list displays only other IMServices
Class[] types = new Class[]{IMService.class};
ServiceTemplate IMTemplate =
   new ServiceTemplate(null, types, null);

// Initialize IMServiceManager, ServiceDiscoveryManager
try {
   myManager = new IMServiceManager(userName);
   // store LookupDiscoveryManager
   // generated by IMServiceManager
   lookupDiscoveryManager = myManager.getDiscoveryManager();
   // ServiceDiscoveryManager uses lookupDiscoveryManager
   serviceDiscoveryManager =
      new ServiceDiscoveryManager(lookupDiscoveryManager, null);
   // create a LookupCache
   cache = serviceDiscoveryManager.createLookupCache(
      IMTemplate, null, this );
}
The methods required to implement `ServiceDiscoveryListener` are `serviceAdded` (lines 50–66), `serviceChanged` (lines 69–70) and `serviceRemoved` (lines 74–98). Method `ServiceAdded` calls method `getPostEventServiceItem` of the `ServiceDiscoveryEvent` to obtain a `ServiceItem` that represents the added service. Lines 57–65 cycle through all the attributes of the `ServiceItem`. If line 59 finds a `Name` entry, line 61 adds the service proxy to a `List`, and lines 62–63 add the service `Name` to a `DefaultListModel`. Method `serviceChanged` is an empty method, because we are not concerned whether services change their attributes. Method `serviceRemoved` invokes method `getPreEventServiceItem` of the `ServiceDiscoveryEvent` to obtain a `ServiceItem` that represents the removed service.

The `ActionListener` (lines 130–167) of the `connectButton` object gets the index of the selected item in the `JList` and retrieves the `IMService` proxy associated with that index from `List serviceItems` (lines 138–141). Lines 147–148 create an `IMPeerListener` object, and lines 149–150 create an `IMPeerImpl` object. Line 151 adds the `IMPeerListener` to the `IMPeerImpl`. The `IMPeerImpl` will post all messages sent by the remote peer to the `IMPeerListener`. Using method `connect` of the `IMService`, line 154 sends a remote reference to the `IMPeerImpl` to the remote peer. Line 155 adds the returned `IMPeer` remote reference to the `IMPeerListener`—this allows the local peer to send messages to remote peer. If this sequence of events throws a `RemoteException`, lines 160–162 inform the user of the error. If the connection succeeds, the service returns an `IMPeer`, and communication occurs as described in earlier sections.
Lines 189–223 create the Add Locator menu item in the File menu. Selecting this menu item launches a dialog box that prompts the user for the URL of a Jini lookup service to add via unicast discovery. Lines 204–213 add this URL to the list of LookupLocators—this results in the client’s registering with the newly added lookup service, and the PeerList window lists all other peers registered with the new lookup service.

### 28.10 Compiling and Running the Example

Finally, we can compile and run the peer-to-peer application. This requires several steps. First, compile the classes using javac.

Next, compile the remote classes IMSServiceImpl and IMPeerImpl using the rmic compiler to produce stub classes (see Chapter 13). Next, place the RMI stub classes that need to be available to other Deitel Instance Messenger clients and lookup services (IMServiceImpl Stub and IMPeerImpl Stub) in a JAR file (e.g., DIM_dl.jar) with the proper package structure.

For lookup services to notify our application when services are added or removed, the ServiceDiscoveryManager needs to upload a remote-event listener. Change to the root directory of your Web server and execute the following command lines:

```shell
jar xvf C:\jini1_1\lib\jini-ext.jar net\jini\lookup\ServiceDiscoveryManager$LookupCacheImpl$LookupListener_Stub.class
jar xvf C:\jini1_1\lib\jini-core.jar net\jini\core\event\RemoteEventListener.class
```

This creates a net subdirectory in your Web server’s root directory.

To use the Deitel Instant Messenger, first start the RMI activation daemon, an HTTP Server, and a lookup service (see Chapter 22). To start Deitel Instant Messenger, change to the directory that contains the application’s package structure and execute the following command line:

```shell
java -Djava.security.policy=policy.all -Djava.rmi.server.codebase=http://host:port/DIM_dl.jar com.deitel.advjhtp1.jini.IM.PeerList
```

Substitute the appropriate values for the host, port and JAR file name (e.g., DIM_dl.jar). The codebase provides the location of the JAR file which contains the RMI stub files that remote peers and lookup services must download.

### 28.11 Improving Deitel Instant Messenger

The Deitel Instant Messenger Jini implementation does not address issues of security and scalability. The ServiceDiscoveryManager downloads proxies for every user listed on every known lookup service. As more users and lookup services join the network, network overhead becomes overwhelming. The lack of security and authentication mechanisms forces anonymity on clients, because there is no absolute way of verifying that users are who they claim to be.
There are various ways to address these problems. Filters can limit the number of service proxies that the ServiceDiscoveryManager downloads. Lookup services can limit the number of services they manage. The application also could use distributed searches to locate peers (see Section 28.4). This involves sending a request to a known search service, which forwards the request if unable to find the user. The ubiquitous use of digital signatures and public and private keys can enable security and authentication—however, this does not solve the problem of having duplicate user names in a large-scale peer-to-peer network. These solutions are incomplete.

### 28.12 Deitel Instant Messenger with Multicast Sockets

The Deitel Instant messenger uses Jini’s JoinManager and ServiceDiscoveryManager to advertise the existence of each peer and discover other peers. With Jini, a few lines of code implemented this functionality. To adhere to a true peer-to-peer architecture, each peer must run a lookup service. Lookup services require significant amounts of memory and processor time. Because of these performance and memory concerns, we introduce an improved implementation of the Deitel Instant Messenger that uses multicast sockets and a simple, text-based protocol to advertise and find peers in the network.

#### 28.12.1 Registering the Peer

In the Jini implementation of the Deitel Instant Messenger, Jini provided the mechanism that allowed peers to sign onto the network and find other peers. If a peer loses its connection without explicitly disconnecting (e.g., by calling method terminate of a JoinManager instance), the Jini lookup services remove the peer after the service’s lease expires. When a peer’s lease expires, the Jini lookup service removes the peer’s service from the registry—in the Deitel Instant Messenger, the peer disappears from the Peer-List window. Because we are removing Jini from this application, we must implement these mechanisms ourselves.

The first mechanism we implement is class MulticastSendingThread (Fig. 28.11). MulticastSendingThread multicasts a peer’s presence. MulticastSendingThread periodically sends multicast packets throughout the network to notify other peers that this peer is still available. Each peer on the receiving end renew’s the lease for the multicasting peer. In the event that the multicasting peer stops advertising its presence (e.g., the user quits the peer application), that peer’s lease expires. Essentially, the peer ceases to exist on the network.

```java
// MulticastSendingThread.java
// Sends a multicast periodically containing a remote reference to the IMServiceImpl object
package com.deitel.advjhtp1.p2p;

// Java core packages
import java.net.MulticastSocket;
import java.net.*;
import java.rmi.*;
```

Fig. 28.11 MulticastSendingThread broadcasts DatagramPackets (part 1 of 4).
import java.rmi.registry.*;
import java.io.*;

// Deitel core packages
import com.deitel.advjhtp1.jini.IM.service.IMServiceImpl;
import com.deitel.advjhtp1.jini.IM.service.IMService;

public class MulticastSendingThread extends Thread
    implements IMConstants {

    // InetAddress of group for messages
    private InetAddress multicastNetAddress;

    // MulticastSocket for multicasting messages
    private MulticastSocket multicastSocket;

    // Datagram packet to be reused
    private DatagramPacket multicastPacket;

    // stub of local peer
    private IMService peerStub;

    // flag for terminating MulticastSendingThread
    private boolean keepSending = true;

    private String userName;

    // MulticastSendingThread constructor
    public MulticastSendingThread( String myName )
    {
        // invoke superclass constructor to name Thread
        super( "MulticastSendingThread" );

        userName = myName;

        // create a registry on default port 1099
        try {
            Registry registry =
                LocateRegistry.createRegistry( 1099 );
            peerStub = new IMServiceImpl( userName );
            registry.rebind( BINDING_NAME, peerStub );
        }
        catch ( RemoteException remoteException ) {
            remoteException.printStackTrace();
        }

        try {
            // create MulticastSocket for sending messages
            multicastSocket =
                new MulticastSocket ( MULTICAST_SENDING_PORT );

Fig. 28.11 MulticastSendingThread broadcasts DatagramPackets (part 2 of 4).
// set TTL for Multicast Socket
multicastSocket.setTimeToLive(MULTICAST_TTL);

// use InetAddress reserved for multicast group
multicastNetAddress = InetAddress.getByName(MULTICAST_ADDRESS);

// create greeting packet
String greeting = new String(HELLO_HEADER + userName);

multicastPacket = new DatagramPacket(greeting.getBytes(), greeting.getBytes().length, multicastNetAddress, MULTICAST_LISTENING_PORT);

// MULTICAST_ADDRESS IS UNKNOWN HOST
catch (java.net.UnknownHostException unknownHostException)
{
    System.err.println("MULTICAST_ADDRESS is unknown");
    unknownHostException.printStackTrace();
}

// any other exception
catch (Exception exception)
{
    exception.printStackTrace();
}

// deliver greeting message to peers
public void run()
{
    while (keepSending) {

        // deliver greeting
        try {

            // send greeting packet
            multicastSocket.send(multicastPacket);

            Thread.sleep(MULTICAST_INTERVAL);
        }

        // handle exception delivering message
        catch (IOException ioException)
        {
            ioException.printStackTrace();
            continue;
        }
        catch (InterruptedException interruptedException)
        {
            interruptedException.printStackTrace();
        }
    }
}

Fig. 28.11 MulticastSendingThread broadcasts DatagramPackets (part 3 of 4).
MulticastSendingThread extends class Thread. Lines 47–48 calls method createRegistry of class LocateRegistry to instantiate an RMI registry on port 1099—the port that rmiregistry application normally uses. Line 49 creates a new IMSServiceImpl object. Line 50 rebinds the IMSServiceImpl object to the RMI registry, using BINDING_NAME—one of many constants defined in interface IMConstants (Fig. 28.12), which class MulticastSendingThread implements. Figure 28.12 shows the interface IMConstants, which defines all the constants used by Deitel Instant Messenger.

Fig. 28.11 MulticastSendingThread broadcasts DatagramPackets (part 4 of 4).

Fig. 28.12 Interface IMConstants defines Deitel-Instant-Messenger constants (part 1 of 2).
Lines 59–60 of Fig. 28.11 create a `MulticastSocket` on the port number that constant `MULTICAST_SENDING_PORT` defines. Line 63 sets the default `Time To Live (TTL)` for `DatagramPacket`s sent through the `MulticastSocket`. Lines 66–67 create an `InetAddress` with the multicast IP address specified by constant `MULTICAST_ADDRESS`.

Lines 70–74 create a `DatagramPacket` that contains a `String` with the peer’s name. `HELLO_HEADER` informs all peers listening on the multicast port that this peer can receive messages. `MULTICAST_LISTENING_PORT` specifies the port on the multicast IP address on which all peers listen. Lines 78–82 catch an `UnknownHostException` if `MULTICAST_ADDRESS` is an invalid multicast IP address.

Line 100 multicasts the `MulticastPacket` that the constructor generated. Line 102 specifies that the thread should wait `MULTICAST_INTERVAL` milliseconds between each multicast. Line 116 closes the `MulticastSocket` when the user exits the application.

Lines 123–127 create a `DatagramPacket` that contains a `String` with the `GOODBYE_HEADER` and the peer’s user name. `GOODBYE_HEADER` indicates that the peer is leaving the network. Lines 130–133 send this `DatagramPacket` and `unbind` the
Chapter 28 Peer-to-Peer Applications and JXTA

IMServiceImpl associated with BNDING_NAME from the RMI registry. Line 146 sets boolean keepSending to false to terminate the thread.

28.12.2 Finding Other Peers

In the Jini implementation of the Deitel Instant Messenger, the Jini lookup service listed new peers and removed peers that left the network. Class ServiceDiscoveryManager updated the application when peers were either added or removed. To implement these mechanisms, we create class MulticastReceivingThread (Figure 28.13), which listens for DatagramPacket that contain notifications of peers joining and leaving the network.

Lines 48–55 create a MulticastSocket on port MULTICAST RECEIVING_PORT and join the multicast group. Line 58 specifies that the MulticastSocket should time out receiving a packet takes longer than five seconds. Lines 69–71 start the LeasingThread as a daemon thread. We explain LeasingThread in greater detail later in this section.

Line 89 receives a DatagramPacket from the MulticastSocket using by invoking method receive. If the call to receive times out, lines 93–97 catch the InterruptedException that is thrown. Lines 106–107 retrieve the message String stored in the received DatagramPacket.

---

1 // MulticastReceivingThread.java
2 // Receive and process multicasts from multicast group
3 package com.deitel.advjhtp1.p2p;
4
5 // Java core packages
6 import java.net.MulticastSocket;
7 import java.net. *;
8 import java.io. *;
9 import java.util. *;
10
11 // Deitel packages
12 import com.deitel.advjhtp1.p2p.PeerDiscoveryListener;
13
14 public class MulticastReceivingThread extends Thread
15   implements IMConstants {
16     // HashMap containing peer names and time to live
17     // used to implement leasing
18     private HashMap peerTTLMap;
19
20     // LeasingThread reference
21     private LeasingThread leasingThread;
22
23     // object that will respond to peer added or removed events
24     private PeerDiscoveryListener peerDiscoveryListener;
25
26 Fig. 28.13 Class MulticastReceivingThread uses threads to add and remove peers (part 1 of 6).
// MulticastSocket for receiving broadcast messages
private MulticastSocket multicastSocket;

// InetAddress of group for messages
private InetAddress multicastNetAddress;

// flag for terminating MulticastReceivingThread
private boolean keepListening = true;

// MulticastReceivingThread constructor
public MulticastReceivingThread( String userName,
   PeerDiscoveryListener peerEventHandler )
{
    // invoke superclass constructor to name Thread
    super( "MulticastReceivingThread" );

    // set peerDiscoveryListener
    peerDiscoveryListener = peerEventHandler;

    // connect MulticastSocket to multicast address and port
    try {
        multicastSocket =
        new MulticastSocket( MULTICAST_RECEIVING_PORT );

        multicastNetAddress =
        InetAddress.getByName( MULTICAST_ADDRESS );

        // join multicast group to receive messages
        multicastSocket.joinGroup( multicastNetAddress );

        // set 5 second time-out when waiting for new packets
        multicastSocket.setSoTimeout( 5000 );
    }

    // handle exception connecting to multicast address
    catch( IOException ioException ) {
        ioException.printStackTrace();
    }

    peerTTLMap = new HashMap();

    // create Leasing thread which decrements TTL of peers
    leasingThread = new LeasingThread();
    leasingThread.setDaemon( true );
    leasingThread.start();

} // end MulticastReceivingThread constructor

// listen for messages from multicast group
public void run()
{

Fig. 28.13 Class MulticastReceivingThread uses threads to add and remove peers (part 2 of 6).
while( keepListening ) {
    // create buffer for incoming message
    byte[] buffer = new byte[ MESSAGE_SIZE ];
    // create DatagramPacket for incoming message
    DatagramPacket packet = new DatagramPacket( buffer,
            MESSAGE_SIZE );
    // receive new DatagramPacket (blocking call)
    try {
        multicastSocket.receive( packet );
    }
    // handle exception when receive times out
    catch ( InterruptedIOException interruptedIOException ) {
        continue;
    }
    // handle exception reading packet from multicast group
    catch ( IOException ioException ) {
        ioException.printStackTrace();
        break;
    }
    // put message data into String
    String message = new String( packet.getData(),
            packet.getOffset(), packet.getLength() );
    // ensure non-null message
    if ( message != null ) {
        // trim extra whitespace from end of message
        message = message.trim();
        System.out.println( message );
        // decide if goodbye or hello
        if ( message.startsWith( HELLO_HEADER ) ) {
            processHello(
                    message.substring( HELLO_HEADER.length() ) ),
                    packet.getAddress().getHostAddress()
                );
        } else if ( message.startsWith( GOODBYE_HEADER ) )
            processGoodbye( message.substring( GOODBYE_HEADER.length() ) );
    } // end if

Fig. 28.13 Class MulticastReceivingThread uses threads to add and remove peers (part 3 of 6).
130     } // end while
131  
132     // leave multicast group and close MulticastSocket
133     try {
134         multicastSocket.leaveGroup( multicastNetAddress );
135         multicastSocket.close();
136     }
137  }
138  
139     // handle exception leaving group
140     catch ( IOException ioException ) {
141         ioException.printStackTrace();
142     }
143 
144 } // end run
145  
146     // process hello message from peer
147     public void processHello( String peerName,  
148         String registryAddress ) 
149     {  
150         registryAddress += ("/" + BINDING_NAME );
151         synchronized( peerTTLMap )
152         {
153             // if it is a new peer, call peerAdded event
154             if ( !peerTTLMap.containsKey( peerName ) ) {
155                 peerDiscoveryListener.peerAdded( peerName,  
156                     registryAddress);  
157             }
158             // add to map or if present, refresh TTL
159             peerTTLMap.put( peerName, new Integer( PEER_TTL ) );
160         }
161     }
162  
163     // process goodbye message from peer
164     public void processGoodbye( String peerName )
165     {
166         synchronized( peerTTLMap )
167         {
168             System.out.println( "Removing peer" + peerName );
169             if ( peerTTLMap.containsKey( peerName ) ) {
170                 peerDiscoveryListener.peerRemoved( peerName );
171                 peerTTLMap.remove( peerName );
172             }
173         }
174     }
175  
176     // periodically decrements the TTL of peers listed
177     private class LeasingThread extends Thread
178     {
179         
Fig. 28.13 Class MulticastReceivingThread uses threads to add and remove peers (part 4 of 6).
public void run()
{
    while ( keepListening )
    {
        // sleep
        try {
            Thread.sleep( MULTICAST_INTERVAL );
        }
        catch ( InterruptedException interruptedException ) {
            interruptedException.printStackTrace();
        }
        // lock hashmap while decrementing TTL values
        synchronized( peerTTLMap ) {
            // decrement peers
            Iterator peerIterator =
                peerTTLMap.entrySet().iterator();
            while ( peerIterator.hasNext() ) {
                // make new TTL of peer
                Map.Entry tempMapEntry =
                    ( Map.Entry ) peerIterator.next();
                Integer tempIntegerTTL =
                    ( Integer ) tempMapEntry.getValue();
                int tempIntTTL = tempIntegerTTL.intValue();
                // decrement TTL
                tempIntTTL--;
                // if lease expired, remove peer
                if ( tempIntTTL < 0 ) {
                    peerDiscoveryListener.peerRemoved(( String ) tempMapEntry.getKey());
                    peerIterator.remove();
                }
                // otherwise set TTL of peer to new value
                else
                    tempMapEntry.setValue(
                        new Integer( tempIntTTL ) );
            } // end while iterating through peers
        } // end synchronized
    } // end while in run method
} // end run method

Fig. 28.13 Class MulticastReceivingThread uses threads to add and remove peers (part 5 of 6).
Lines 118–123 call method `processHello` (lines 147–164) if the message `String` begins with constant `HELLO_HEADER`. Lines 125–127 call method `processGoodbye` (lines 167–177) if the message starts with `GOODBYE_HEADER`. When the `while` loop terminates, lines 134–187 unsubscribe from the multicast group and close the socket.

Method `processHello` handles messages that contain the `HELLO_HEADER`. Line 150 appends `BINDING_NAME` to the given `String` containing the IP address of the peer that sent the “hello” message. This forms an RMI URL with which the peer can connect to the newly joined peer. Line 151 synchronizes object `HashMap peerTTLMap`, to prevent other threads from accessing and modifying `peerTTLMap`. This `HashMap` stores peer names as keys and stores the peer lease-expiration times as values. Line 155 tests if the peer name specified in the “hello” message is in `peerTTLMap`. If the peer is not in `peerTTLMap`, line 156 calls method `peerAdded` of the `PeerDiscoveryListener` object. Each `MulticastReceivingThread` contains a reference of an object that implements interface `PeerDiscoveryListener` (Fig. 28.14).

Methods `peerAdded` (line 8) and `peerRemoved` (line 11) inform the `PeerDiscoveryListener` implementation that a peer has entered, or has been removed from, the multicast group, respectively. Line 161 of class `MulticastReceivingThread` puts an entry into `peerTTLMap` that contains the peer’s name and TTL (specified by `PEER_TTL`). If the peer is already in `peerTTLMap`, line 161 replaces the pre-existing entry with a new entry, which renews the peer’s lease by resetting its Time to Live (TTL). Line 174 of class `MulticastReceivingThread` removes a peer from `peerTTLMap`. Because every peer continuously multicasts “hello” packets, datagram packets may be duplicated. Therefore, line 172 first checks if the given peer is in `peerTTLMap` before attempting to remove it. Line 173 calls method `peerRemoved` on the registered `PeerDiscoveryListener` to inform it that a peer has left the multicast group.

```
234 } // end class LeasingThread
235
236 // stop listening for multicasts
237 public void logout()
238 {
239     // terminate thread
240     keepListening = false;
241 }
242 }
```

Fig. 28.13 Class `MulticastReceivingThread` uses threads to add and remove peers (part 6 of 6).

```
1 // PeerDiscoveryListener.java
2 // Interface for listening to peerAdded or peerRemoved events
3 package com.deitel.advjhtp1.p2p;
4
5 public interface PeerDiscoveryListener {
6
```

Fig. 28.14 Interface `PeerDiscoveryListener` listens for when peers are added and removed from peer groups (part 1 of 2).
Lines 180–235 define inner class `LeasingThread`. A single-leasing-thread instance periodically decrements the TTL of each peer in the `peerTTLMap`. Line 197 synchronizes `peerTTLMap` to prevent modification conflicts between threads trying to access `peerTTLMap` concurrently. Lines 200–201 obtain an `Iterator` object of `peerTTLMap`'s entries. Lines 203–227 decrement the TTL of each entry in `peerTTLMap`. Lines 208–210 obtains an `int` that contains a peer’s TTL—line 213 decrements the TTL value. Line 216 checks if the modified TTL is less than zero, which indicates that the peer’s lease has expired. Lines 217–218 then pass the expired peer’s name as an argument to method `peerRemoved` of the `PeerDiscoveryListener` object. Line 219 calls `Iterator` method `remove`, which removes the current entry from `peerTTLMap`. If the modified TTL is greater than (or equal to) zero, the peer’s lease is still active, so lines 224–225 set the peer’s TTL to the newly decremented value. Method `logout` (lines 238–242) enables an outside object to terminate the `MulticastReceivingThread`.

To use classes `MulticastReceivingThread` and `PeerDiscoveryListener` in the Deitel Instant Messenger, we had to modify class `PeerList` from our Instant Messenger Jini implementation. Figure 28.15 contains the modified `PeerList` listing.
import com.deitel.advjhtp1.jini.IM.service.IMService;
import com.deitel.advjhtp1.jini.IM.client.IMPeerListener;
import com.deitel.advjhtp1.jini.IM.IMPeerImpl;
import com.deitel.advjhtp1.jini.IM.IMPeer;

public class PeerList extends JFrame
    implements PeerDiscoveryListener, IMConstants {

    // initialize userName to anonymous
    private String userName = "anonymous";
    private MulticastSendingThread multicastSender;
    private MulticastReceivingThread multicastReceiver;

    // list variables
    private DefaultListModel peerNames;   // contains peer names
    private List peerStubAddresses;       // contains peer stubs
    private JList peerJList;

    // add peer name and peer stub to lists
    public void peerAdded( String name, String peerStubAddress )
    {
        // add name to peerNames
        peerNames.addElement( name );

        // add stub to peerStubAddresses
        peerStubAddresses.add( peerStubAddress );
    } // end method peerAdded

    // removes services from PeerList GUI and data structure
    public void peerRemoved( String name )
    {
        // remove name from peerNames
        int index = peerNames.indexOf( name );
        peerNames.removeElementAt( index );

        // remove stub from peerStubAddresses
        peerStubAddresses.remove( index );
    } // end method peerRemoved

    // constructor
    public PeerList()
    {
        super( "Peer List" );

        // get desired userName
        userName = JOptionPane.showInputDialog(
            PeerList.this, "Please enter your name: ");
    }

Fig. 28.15 Modified PeerList enables the use of classes
MulticastReceivingThread and PeerDiscoveryListener
in the Deitel Instant Messenger (part 2 of 6).
Fig. 28.15 Modified PeerList enables the use of classes MulticastReceivingThread and PeerDiscoveryListener in the Deitel Instant Messenger (part 3 of 6).
// malformedURL passed to lookup
   catch ( MalformedURLException exception ) {
      JOptionPane.showMessageDialog
      ( null, "Stub address incorrectly formatted" );
      exception.printStackTrace();
   }
   
   // Remote object not bound to remote registry
   catch ( NotBoundException notBoundException ) {
      JOptionPane.showMessageDialog
      ( null, "Remote object not present in Registry" );
      notBoundException.printStackTrace();
   }
   
   // connecting may cause RemoteException
   catch ( RemoteException remoteException ) {
      JOptionPane.showMessageDialog
      ( null, "Couldn't Connect" );
      remoteException.printStackTrace();
   }

   } // end method ActionPerformed
   
   } // end ActionListener anonymous inner class
   
   }); // end connectButton actionListener
   
   // set up File menu
   JMenu fileMenu = new JMenu( "File" );
   fileMenu.setMnemonic( 'F' );
   
   // about Item
   JMenuItem aboutItem = new JMenuItem( "About..." );
   aboutItem.setMnemonic( 'A' );
   aboutItem.addActionListener(
      new ActionListener() {
         public void actionPerformed( ActionEvent event ) {
            JOptionPane.showMessageDialog( PeerList.this,
            "Deitel Instant Messenger",
            "About", JOptionPane.PLAIN_MESSAGE );
         }
      }
   );
   
   // set up JMenuBar and attach File menu
   JMenuBar menuBar = new JMenuBar();
   menuBar.add ( fileMenu );

Fig. 28.15 Modified PeerList enables the use of classes
MulticastReceivingThread and PeerDiscoveryListener
in the Deitel Instant Messenger (part 4 of 6).
setJMenuBar( menuBar );

// handow window closing event
addWindowListener(
    new WindowAdapter(){
        public void windowClosing( WindowEvent w )
        {
            System.out.println( "CLOSING WINDOW" );

            // disconnects from lookup services
            multicastSender.logout();
            multicastReceiver.logout();

            // join threads
            try {
                multicastSender.join();
                multicastReceiver.join();
            }
            catch( InterruptedException interruptedException ) {
                interruptedException.printStackTrace();
            }

            System.exit( 0 );
        }
    });

// lay out GUI components
peerJList.setFixedCellWidth( 100 );
JPanel inputPanel = new JPanel();
inputPanel.add( connectButton );
container.add( new JScrollPane( peerJList ) ,
    BorderLayout.NORTH );
container.add( inputPanel, BorderLayout.SOUTH );

// Initialize threads
try {

    multicastReceiver =
        new MulticastReceivingThread( userName, this );
    multicastReceiver.start();

    multicastSender =
        new MulticastSendingThread( userName );
    multicastSender.start();

}
Class **PeerList** implements interface **PeerDiscoveryListener**—the Jini version implemented interface **ServiceDiscoveryListener**. Lines 210–211 create a **MulticastReceivingThread** and passes the this reference as a **PeerDiscoveryListener**—line 212 starts this thread. Lines 214–215 create a **MulticastSendingThread** specifying the user name—line 216 starts this thread.

Lines 39–47 implement method **PeerAdded**, which takes two **Strings**—**name** and **peerStubAddress**. Parameter **name** specifies the peer’s name. Parameter **peerStubAddress** is a URL-formatted **String** that contains the information needed to make a **Naming.lookup** call on the remote peer. The **Naming.lookup** call obtains an **IMService** remote reference. Lines 42–45 store the **IMService** remote reference.

Lines 51–60 implement method **peerRemoved**. Lines 54–58 remove the information of the given peer from **peerNames** and **peerStubAddresses**.

Lines 89–141 specify the **ActionListener** for the **JButton** that we use to connect peers. Line 96–97 obtains the **peerStubAddress** of the selected peer in the **JList**. Lines 102–104 calls method **Naming.lookup** to obtain a reference to the remote peer’s **IMService** object from the RMI registry. The code in lines 106–140 works similarly to that of the previous Deitel Instant Messenger implementation.

Lines 176–194 specify instructions to execute when the user closes the **PeerList** window. Lines 180–181 terminate each thread by calling method **logout**. Lines 186–187 **join** each thread, thereby blocking until each thread terminates. Line 193 exits the program.

### 28.13 Introduction to JXTA

Sun Microsystems, Inc. created Project JXTA\(^3\) as a response to the growing popularity of peer-to-peer applications. Project JXTA strives to create a standard, low-level, platform and language-independent protocol that promotes interoperability among peer-to-peer applications. The current JXTA implementation is written in Java, but developers can imple-

---

3. For more information, see [www.jxta.org](http://www.jxta.org).
ment JXTA in any programming language. JXTA provides a foundation from which developers can build any type of P2P application.

JXTA attempts to solve the following problems of peer-to-peer applications:

1. Security/Authentication—Large peer-to-peer network applications, such as AOL Instant Messenger and MSN Instant Messenger, use central servers to bootstrap users onto the network. This bootstrapping ensures, to some degree, that the same person uses a particular online identity.

2. Peer Discovery—Without a central server, it is difficult to realize the presence of other peers on the network. Multicasting, as used by Jini, is not a viable solution outside the LAN setting.

3. Network Incompatibility—Currently, each popular peer-to-peer application yields a set of proprietary protocols that prevent compatibility with other peer-to-peer networks. For example, the millions of users on the AIM platform cannot communicate with Yahoo Instant Messenger users. Most new users opt for the peer-to-peer application with the largest following.

4. Platform Incompatibility—Software developers must rewrite the low-level core aspects of their peer-to-peer applications for each platform they wish to support. Wireless phones and other mobile devices usually have a limited selection of P2P applications, if any.

JXTA attempts to solve these problems by standardizing the low-level protocols that govern peer-to-peer applications. JXTA is designed to be a general infrastructure, rather than a special-purpose one. Therefore, developers can use JXTA to implement virtually any type of P2P application. Because all JXTA-based P2P applications use identical low-level protocols, they will be compatible with one another.

Networks built with the JXTA protocols consist of three basic types of entities—peer/peer groups, advertisements and pipes/messages. Each JXTA runtime environment associates each entity’s name and network address with a unique 128-bit identifier.

A peer is any entity that uses JXTA protocols (Fig. 28.16) to communicate with other peers. Each peer need support only some of the protocols, so devices with low processing power and memory can participate in JXTA networks (albeit with limited functionality). Peer groups are logical constructs that represent sets of peers. JXTA specifies only two rules regarding peer groups:

1. peers can join or leave groups
2. the group administrator, if the group has one, controls access to the group.

All peers are part of the World Peer Group. Membership in the World Peer Group does not imply that each peer can discover and communicate with every other peer on the network.

Advertisements are XML documents that perform a function similar to that of multicast packets in Jini. An entity in the JXTA network advertises itself to notify others of its existence by sending XML documents formatted according to JXTA specifications.

At the simplest level, pipes are unreliable one-way communication channels. More sophisticated pipes may be reliable and multi-directional. Earlier in the chapter, we mentioned that an RMI peer reference allows one-way communication with that peer. Pipes perform similarly. Two peers communicate by configuring two pipes that “flow” in opposite directions. Each peer communicates by sending Messages to the other peer through the
JXTA specifies the Messages’ structure. The most recent JXTA implementation uses XML Messages. The developers of JXTA use XML because of its portability. However, JXTA does not restrict the Message format to XML.

JXTA is still in development and has not yet resolved all of the problems with peer-to-peer applications. The implementation of peer discovery and security measures continues to progress. JXTA suggests that peer discovery use a combination of LAN-based discovery, discovery through invitation, cascaded discovery and rendezvous discovery. Jini illustrates one means of LAN-based discovery. In LAN-based discovery, peers in a local network discover each other automatically by multicasting. Discovery through invitation occurs when a peer receives a message from a previously unknown peer. Cascaded discovery is a distributed-search mechanism—similar to that of Gnutella. Rendezvous discovery is the creation of certain well-known sites that provide network addresses of many users on the network to bootstrap new users.

For further information on the current state of JXTA and other P2P technologies, see the resources provided in Section 28.14.

### 28.14 Internet and World Wide Web Resources

**www.openp2p.com**

openp2p.com is a Web site that is part of the O’Reilly Network. This online resource provides articles and links about peer-to-peer technologies.

**www.clip2.com**

This site provides information and statistics regarding the popular and upcoming peer-to-peer technologies. This site also provides columns that explain how various peer-to-peer protocols work.

**www.peer-to-peerwg.org**

This page is published by the peer-to-peer working group.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer Discovery</td>
<td>Peers use this protocol to find other entities in the JXTA network by searching for advertisements.</td>
</tr>
<tr>
<td>Peer Resolver</td>
<td>Peers that help a search process (e.g., higher bandwidth, capacity storage, etc.) implement this protocol.</td>
</tr>
<tr>
<td>Peer Information</td>
<td>Peers obtain information about other peers via this protocol.</td>
</tr>
<tr>
<td>Peer Membership</td>
<td>Peers use this protocol to learn about the requirements of groups, how to apply for membership, how to modify their membership and how to quit a group. Authentication and security are implemented through this protocol.</td>
</tr>
<tr>
<td>Pipe Binding</td>
<td>Peers can connect pipes to one another, via advertisements, through this protocol.</td>
</tr>
<tr>
<td>Endpoint Routing</td>
<td>Peer routers implement this protocol to provide other routing services to other peers (e.g., tunneling through a firewall).</td>
</tr>
</tbody>
</table>

Fig. 28.16 JXTA low-level protocols.
www.jxta.org
This is the official Web site for Project JXTA and contains the newest downloads of the source code and opportunities to participate in developing JXTA.

www.peerintelligence.com
This Web site publishes columns that discuss how peer-to-peer technologies are being developed into enterprise solutions. This site focuses on how businesses can use peer-to-peer applications.

www.peertal.com
This site presents columns that discuss peer-to-peer technologies.

**SUMMARY**

- In a peer-to-peer (P2P) network architecture, each node can perform both client and server functions. Such networks distribute processing and information to many computers, thus reclaiming otherwise wasted computing power and storage space and eliminating central points of failure.
- Developers may implement P2P applications using various technologies, such as multicast sockets.
- Many networks operate on the principle that computers should be segregated by function.
- Instead of segregating computers by function, in P2P networks all computers act as both clients and servers.
- The concept of P2P applications is similar to that of the telephone system—a single user can both speak (send information) and listen (receive information).
- Many network applications do not fall neatly into either client/server or peer-to-peer categories.
- One major weakness of this centralized system is the dependency on the central server. If the central node (i.e., server) fails, so does the entire application.
- The capabilities of the server limit the overall performance of the application.
- Centralized architectures simplify management tasks, such as monitoring user access, by providing single points of network control.
- True P2P applications are completely decentralized and do not suffer from the same deficiencies as applications that depend on centralized servers.
- Some P2P applications leverage distributed computational power.
- Peer-to-peer architectures allow real-time searches that return up-to-date results.
- Peer-to-peer searches reflect the status of the network at the time of the query.
- In peer-to-peer networks determining who is on the network at any instant is difficult.
- Real-time searches are slower and increase network traffic, because every query must propagate throughout the entire network.
- A true client/server network is completely centralized, whereas a true peer-to-peer application is completely decentralized.
- Many applications adopt aspects of both networking styles to achieve specific goals.
- Peer discovery is the act of finding peers in a P2P application.
- Decentralizing an application makes peer discovery and searching for information difficult.
- Distributed searches make networks more robust by removing single points of failure, such as central servers.
- Information found via distributed searches is up-to-date because it reflects the current state of the network.
TERMINOLOGY

authentication  
bootstrapping  
central server  
centralization  
centralization  
client/server computing  
decentralization  
distributed search  
Freenet  
Gnutella  
Jini  
JXTA  
lookup service  
multicast socket  

P2P (peer-to-peer) application  
peer  
peer discovery  
peer group  
pipe  
proxy  
real-time search  
Remote Method Invocation (RMI)  
search engine  
search hub  
Time to Live (TTL)  
unicast discovery  

SELF-REVIEW EXERCISES

28.1 Fill in the blanks in each of the following statements:
   a) For the Deitel Instant Messenger to send a message to a remote peer, it must hold a _______ to the remote peer.
   b) Pipes are _______ one-way communication channels.
   c) Large peer-to-peer networks, such as AOL Instant Messenger and MSN Instant Messenger, use _______ to bootstrap users onto the network.
   d) Peer discovery is the act of _______ and _______ with other peers.
   e) Class _______ can be used to cache proxies of all services listed in known lookups services.

28.2 State whether each of the following is true or false. If false, explain why.
   a) In a peer-to-peer application, each peer performs both client and server functions.
   b) One remote reference provides one-way communication.
   c) Jini is the best tool for designing peer-to-peer applications.
   d) Client/Server network architecture is the most efficient way of organizing groups of computers.
   e) The ServiceDiscoveryManager needs to export certain classes to asynchronously notify ServiceDiscoveryListeners of ServiceDiscoveryEvents.
   f) Broadcasting is a viable solution to the problem of discovering other peers on a large-scale peer-to-peer network.
   g) All network applications are either peer-to-peer or client/server.

ANSWERS TO SELF-REVIEW EXERCISES

28.1 a) reference.  b) unreliable.  c) central servers.  d) finding, connecting.  e) ServiceDiscoveryManager.
28.2 a) True.  b) True.  c) False. Implementing authentication, security and large-scale networks is difficult in Jini. There is no automatic means of peer discovery by distributed search.
   d) False. Client/server computing wastes the unused processing power and storage space of the client computers.
   e) True.
f) False. Broadcasting is an inefficient way for computers to announce their existence in a large network. As the number of computers in a network grow, broadcasting creates too much network traffic to be a viable solution.


g) False. Many applications include elements of both peer-to-peer applications and client/server applications.

**EXERCISES**

**28.3** Modify Deitel Instant Messenger so that it can be used to send image files and display them on the remote computer.

**28.4** Modify Deitel Instant Messenger so that it plays a user-selected sound file when an incoming message is received.

**28.5** Deitel Instant Messenger can create multiple conversations with the same person. Modify Deitel Instant Messenger so that it will not allow multiple conversations with the same person simultaneously.

**28.6** Modify Deitel Instant Messenger so that peers notify any peers with whom they are speaking when they close the conversation window. Remote peers will reconnect to continue sending messages.

**28.7** Expand Deitel Instant Messenger so that users can have a user profile. Enable users to search for users with keywords.
Introduction to Web Services and SOAP

Objectives

- To understand the Simple Object Access Protocol (SOAP) and how it uses XML.
- To understand the structure of a SOAP message.
- To be able to write Java applications that send and receive SOAP messages.

Nothing happens until something is sold.
Arthur H. Motley

Men are going to have to learn to be managers in a world where the organization will come close to consisting of all chiefs and one Indian. The Indian, of course, is the computer.
Thomas L. Whisler

…it is always the simple that produces the marvelous.
Amelia Barr

Resemblance reproduces the formal aspect of objects...
Ching Hao

…if the simple things of nature have a message that you understand, rejoice, for your soul is alive...
Eleonora Duse
Interoperability, or seamless communication and interaction between different software systems, is a primary goal of many businesses and organizations that rely heavily on computers and electronic networks. Many applications use the Internet to transfer data. Some of these applications run on client systems with little processing power, so they invoke method calls on remote machines to process data. Many applications use proprietary data specifications, which makes communication with other applications difficult, if not impossible. The majority of applications also reside behind firewalls—security barriers that restrict communication between networks. The Simple Object Access Protocol (SOAP) is a protocol that addresses these problems. Combining the powers of HTTP and XML, it provides a fully extensible mode of communication between software systems.

Web Services are an emerging area of distributed computing. Sun Microsystems’ Open Net Environment (ONE) and Microsoft Corporation’s .NET initiative are frameworks for writing and deploying Web services. There are several definitions of a Web service. A Web service can be any Web-accessible application, such as a Web page with dynamic content. A narrower definition of a Web service is an application that exposes a public interface usable by other applications over the Web. Sun’s ONE requires Web services to be accessible through HTTP and other Web protocols, to communicate with XML-based messages and to be available through lookup services. SOAP provides the XML communication in many Web services. Web services can provide great interoperability between diverse systems.1

A hypothetical Web service designed for the Sun ONE architecture might take a form in which a service registry publishes a description of the Web service as a Universal Description, Discovery and Integration (UDDI) document. The client, such as a Web browser or Java GUI client, searches a directory service for a needed Web service. The client uses the information it receives through the lookup service to send an XML message via HTTP to the Web server hosting the Web service. A servlet processes the client request. The servlet then accesses an application server that provides Enterprise Java Beans. The EJBs in turn access a database that stores the Web service’s information. After accessing the database, the EJB responds to the servlet with the requested information. The servlet formats the information for the client (e.g. creates a JavaServer Page). The HTTP server sends an XML response back to the client. The client then parses the response and displays the information for the user.1

The great potential of Web services does not lie with the technology used to create them. HTTP, XML and the other protocols used by Web services are not new. The interoperability and scalability of Web services means developers can rapidly create large applications and larger Web services from small Web services. Sun’s Open Net Environment describes an architecture for creating smart Web services. Smart Web services share a common operating environment with one another. By sharing context, smart Web services can perform common
authentication for financial transactions, provide location-specific recommendations and directions, etc., among e-businesses. At the time of this writing, there are two major obstacles in the way of developing smart Web services. There do not yet exist widely accepted standards for sharing context between Web services or ensuring the security and privacy of Web-service transactions.

29.2 Simple Object Access Protocol (SOAP)

IBM, Lotus Development Corporation, Microsoft, DevelopMentor and Userland Software developed and drafted SOAP, which is an HTTP-XML-based protocol that enables applications to communicate over the Internet, by using XML documents called SOAP messages. SOAP is compatible with any object model, because it includes only functions and capabilities that are absolutely necessary for defining a communication framework. Thus, SOAP is both platform and software independent, and any programming language can implement it. SOAP supports transport using almost any conceivable protocol. For example, SOAP binds to HTTP and follows the HTTP request–response model. SOAP also supports any method of encoding data, which enables SOAP-based applications to send virtually any type information (e.g., images, objects, documents, etc.) in SOAP messages.

A SOAP message contains an envelope, which describes the content, intended recipient and processing requirements of a message. The optional header element of a SOAP message provides processing instructions for applications that receive the message. For example, for implementations that support transactions, the header could specify details of that transaction. The header also can incorporate routing information. Through the header, more complex protocols can be built onto SOAP. Header entries can modularly extend the message for purposes such as authentication, transaction management and payment. The body of a SOAP message contains application-specific data for the intended recipient of the message.

SOAP has the ability to make a Remote Procedure Call (RPC), which is a request made to another machine to run a task. The RPC uses an XML vocabulary to specify the method to be invoked, any parameters the method takes and the Universal Resource Identifier (URI) of the target object. An RPC call naturally maps to an HTTP request, so an HTTP post sends the message. A SOAP-response message is an HTTP-response document that contains the results of the method call (e.g., returned values, error messages, etc.). SOAP also supports asynchronous RPC, in which program that invokes the RPC does not wait for a response from the remote procedure.

At the time of this writing, SOAP is still under development, and many of the technologies that build on it are in the early stages of development. To realize the benefits of SOAP, industry must establish higher-level specifications and standards that use this technology. Nevertheless, SOAP is the leading industry standard for an XML-distributed computing infrastructure, providing previously nonexistent extensibility and interoperability.

Figure 29.1–Fig. 29.4 present a SOAP example using Apache’s SOAP implementation API, version 2.2 (available at xml.apache.org/soap). The SOAP RPC requires a servlet engine, such as Tomcat (jakarta.apache.org) and Apache’s Xerces parser for Java (available at xml.apache.org/xerces-j/index.html). The SOAP documentation (docs/install/index.html) includes installation instructions for both the server and the client.

Figure 29.1 shows class SimpleService, which resides on the server and contains method getWelcome. The Java application in Fig. 29.4 invokes this method using an RPC.
Method \texttt{getWelcome} (lines 6–12) returns a \texttt{String} when invoked. To make this method available to clients (i.e., to facilitate RPC), we need to provide the server with the name of the method that processes the request—i.e., we must \textit{deploy the service}.

To deploy the service, first copy the \texttt{SimpleService.class} file into the \texttt{jakarta-tomcat/classes} directory. If you created a Java Archive (JAR) file, copy the JAR file into the \texttt{jakarta-tomcat/lib} directory. Create the \texttt{classes} or \texttt{lib} directories if they do not exist. Jakarta-Tomcat includes files in these directories in the \texttt{CLASSPATH}.

Deploy the service with the XML-SOAP administration tool included in the SOAP package (located in the directory \texttt{webapps/soap}). To run this application, type the URL 
\texttt{localhost:8080/soap/admin} into a Web browser. Figure 29.2 and Fig. 29.3 show the administration tool that allows you to deploy, remove and list services. The \texttt{ID field} in Fig. 29.2 contains a URI (\texttt{urn:xml-simple-message}) that identifies the service to the client. This URI is programmer defined. If one service has the same URI as another, the client cannot differentiate between them; consequently, errors may occur. The \texttt{Scope field} specifies the lifetime of the object created (on the server) for serving the SOAP request. The object can exist for the duration of the \texttt{Request, Session} or \texttt{Application}. \texttt{Request} denotes that the server deletes the object after it sends the response, \texttt{Session} indicates that the object persists for the duration of the client’s session with the server. \texttt{Application} signifies that the object is available for all requests throughout the lifetime of the application. The \texttt{Methods field} (Fig. 29.2) specifies the methods available for a SOAP request—in this case, method \texttt{getWelcome}. The \texttt{Provider Type field} specifies the service implementation language. Languages supported include Java, JavaScript, Perl and Bean Markup Language (BML). For the examples in this chapter, we use Java. In the \texttt{Provider Class field} we specify the class that implements the service—\texttt{SimpleService}. The \texttt{Script Language, Script File} and \texttt{Script} fields are used only for services implemented in a supported scripting language. The \texttt{Type Mapping} field allows manual mapping of Java types to XML. The Apache SOAP implementation provides default mappings for most Java types and for Java classes that follow the JavaBeans design patterns. After completing the form, click the \texttt{Deploy} button on the bottom of the form to deploy the service. Click the \texttt{List} button, which lists the services, to confirm that the service deployed successfully (Fig. 29.3). Instructions for other methods of deployment (such as using the command line) are provided in \texttt{docs\guide\index.html}.

\begin{verbatim}
// Fig. 29.1: SimpleService.java
// Implementation for the requested method on the server

public class SimpleService {

    public String getWelcome( String message ) throws Exception {
        String text = "Welcome to SOAP!\nHere is your message: " + message;
        return text;  // response
    }
}

Fig. 29.1 Class SimpleService.
\end{verbatim}
Figure 29.4 lists the client-side code for the RPC. When executed, the program sends a SOAP request to the server, which in our case is the same machine, the local host. The client sends a message as a parameter to the remote method. (This message can be from the command line; by default, the application uses the message **Thanks!**) When the server invokes the method, it sends back the message

**Welcome to SOAP!**
**Here is your message: Thanks!**
Fig. 29.3  Description of deployed service.

```java
// Fig. 29.4 : GetMessage.java
// Program that makes a SOAP RPC

// import Java packages
import java.io.*;
import java.net.*;
import java.util.*;

// import third-party packages
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class GetMessage {

    public static void main( String args[] ) {
        String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
        String message;
    }
}
```
```
if ( args.length != 0 )
    message = args[ 0 ];
else
    message = "Thanks!";

// attempt SOAP remote procedure call
try {
    URL url = new URL(
        "http://localhost:8080/soap/servlet/rpcrouter" );

    // build call
    Call remoteMethod = new Call();
    remoteMethod.setTargetObjectURI( "urn:xml-simple-message" );

    // set name of remote method to be invoked
    remoteMethod.setMethodName( "getWelcome" );
    remoteMethod.setEncodingStyleURI( encodingStyleURI );

    // set parameters for remote method
    Vector parameters = new Vector();
    parameters.addElement( new Parameter( "message", String.class, message, null ) );
    remoteMethod.setParams( parameters );
    Response response;

    // invoke remote method
    response = remoteMethod.invoke( url, "" );

    // get response
    if ( response.generatedFault() ) {
        Fault fault = response.getFault();

        System.err.println( "CALL FAILED:
            Fault Code = "
            + fault.getFaultCode()+ "\nFault String = "
            + fault.getFaultString() );
    } else {
        Parameter result = response.getReturnValue();

        // display result of call
        System.out.println( result.getValue() );
    }
}

// catch malformed URL exception
catch ( MalformedURLException malformedURLException ) {
    malformedURLException.printStackTrace();
    System.exit( 1 );
}
```

Fig. 29.4  Client making a SOAP request (part 2 of 3).
Line 10 imports the SOAP package that provides the API for the SOAP implementation. The package `org.apache.soap.rpc` in line 11 provides the implementation for RPC using SOAP. Line 17 specifies the encoding style used for the message. SOAP, which has no default encoding style, supports many encoding styles—we use the standard RPC encoding (`WS_URI_SOAP_ENC`). Lines 27–28 specify the server-side URL to which the client sends message’s value, `rpcrouter`. This document, a Java servlet, receives the SOAP envelope through the HTTP `post` method. Using the URI specified in the SOAP message, it looks up the services deployed on the server in order to instantiate the appropriate object, in this case a `SimpleService` object.

Objects of class `Call` invoke remote methods. Line 31 instantiates a `Call` object and assigns it to reference `remoteMethod`, and lines 32–33 set the remote method’s URI. Line 36 specifies the name of the method to be invoked, `getWelcome`. We then set the encoding style for the message on line 37. Lines 40–44 build the parameters passed to the remote method for processing. Each parameter must be in its own object, and the parameter objects must be placed in a `Vector`.

Lines 42–43 build a new parameter for the method by constructing a `Parameter` object. The first constructor argument is the name of the variable or reference (message), the second argument is the class to which the `Parameter` object belongs (`String`), the third argument is the value of the parameter (the object `message`) and the fourth argument specifies the parameter’s encoding. (null specifies the application’s default encoding.) Method `setParams` in line 44 sets the parameters of the `remoteMethod` object.

We invoke the remote method by calling method `invoke` in line 48. It takes two arguments: the server URL to which the SOAP message is being sent and the value of the `SOAPAction` header, which specifies the intent of the request. The second argument can take a null string if a `SOAPAction` header is not being used. Method `invoke` throws a `SOAPException` (lines 74–78) if any network error occurs while the SOAP request is being sent. Once the method is invoked on the server, the result is sent back to the client.
and stored in the object referenced by `response` (line 48). This object receives an error message if a server error, such as a failure to locate the appropriate services, occurs. Lines 51–57 determine whether the received message is an error message. Lines 59–64 print the output if no error has been received.

### 29.3 SOAP Weather Service

This section describes a simple Web service implemented with Java and SOAP. The Web service is a modified version of the weather service implemented with RMI in Chapter 13. The weather service now uses SOAP RPC instead of Java RMI to send information from the server to the client. The required software components are the same as the previous section.

The SOAP weather service uses classes `WeatherBean`, `WeatherCellRenderer`, `WeatherItem` and `WeatherListModel` from the RMI weather service without modification. SOAP RPC does not require a Java interface like RMI, so this example does not use interface `WeatherService` and class `WeatherServiceImpl`. For this example, class `WeatherService` uses most of `WeatherServiceImpl`’s code and exposes method `getWeatherInformation` via SOAP. Class `WeatherServiceClient` now uses SOAP RPC instead of Java RMI.

Class `WeatherService` (Fig. 29.5) provides method `getWeatherInformation` that class `WeatherServiceClient` calls through a SOAP RPC. `WeatherService` must be in the `classes` directory of the Tomcat servlet engine for the call to be successful. The RMI version of method `getWeatherInformation` returns a `List` of `WeatherBean` objects. SOAP does not support direct transmission of these Java objects, so the SOAP RPC version of `getWeatherInformation` returns a `Vector` of `Strings`. Lines 66–71 add the `Strings` parsed from the `Traveler’s Forecast` Web page to `Vector weatherInformation` (line 13). Line 99 returns `Vector weatherInformation`. The rest of class `WeatherService` is identical to class `WeatherServiceImpl` from Chapter 13.

```java
// Fig. 29.5: WeatherService.java
// WeatherService provides a method to retrieve weather information from the National Weather Service.
package com.deitel.advjhtp1.soap.weather;

import java.io.*;
import java.net.URL;
import java.util.*;

public class WeatherService {

    private Vector weatherInformation;  // WeatherBean objects

    // get weather information from NWS
    private void updateWeatherConditions() {
        try {
            System.out.println( "Update weather information..." );
```
// National Weather Service Travelers Forecast page
URL url = new URL("http://iwin.nws.noaa.gov/iwin/us/traveler.html");

// set up text input stream to read Web page contents
BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));

// helps determine starting point of data on Web page
String separator = "TAV12";

// locate separator string in Web page
while (!in.readLine().startsWith(separator)) {} // do nothing

// strings representing headers on Travelers Forecast Web page for daytime and nighttime weather
String dayHeader = "CITY            WEA     HI/LO   WEA     HI/LO";
String nightHeader = "CITY            WEA     LO/HI   WEA     LO/HI";

String inputLine = "";

// locate header that begins weather information
do {
inLine = in.readLine();
} while (!inputLine.equals(dayHeader) && !inputLine.equals(nightHeader));

weatherInformation = new Vector(); // create Vector

// create WeatherBeans containing weather data and store in weatherInformation Vector
inputLine = in.readLine(); // get first city's data

// The portion of inputLine containing relevant data
// is 28 characters long. If the line length is not at least 28 characters long, then done processing data.
while (inputLine.length() > 28) {

    // Prepare strings for WeatherBean for each city.
    // First 16 characters are city name. Next, six characters are weather description. Next six
    // characters are HI/LO or LO/HI temperature.
    weatherInformation.add(inputLine.substring(0, 16));
    weatherInformation.add(inputLine.substring(16, 22));
    weatherInformation.add(inputLine.substring(23, 29));
}

Fig. 29.5 SOAP implementation of class WeatherService (part 2 of 3).
Class WeatherServiceClient (Fig. 29.6) makes a SOAP remote procedure call to method getWeatherInformation of class WeatherService. Lines 14–15 import the Apache SOAP packages. Lines 31–32 set the SOAP service’s URL. Line 35 creates a new Call object, which stores information needed to perform the remote procedure call. Lines 36–37 set the URI that uniquely identifies the weather service in the servlet engine. Lines 40–41 specify the method name for the RPC. Lines 42–43 set the encoding for the call. Line 46 creates a Response object and calls method invoke on the Call object with a URL as an argument. The Response object holds the response to the remote procedure call. Line 49 determines if the response generates a Fault. If a Fault occurs, lines 52–54 print the error code. If no Fault occurs, line 58 gets the object returned by the remote procedure call. Lines 60–61 cast the Object to a Vector. Lines 64–65 create a List and call method createBeans (lines 95–107) with the Vector of Strings as a parameter. Method createBeans turns the Vector of Strings into a List of WeatherBeans. Lines 68–69 create a ListModel of the List of WeatherBeans. Lines 73–77 create a JList of the information obtained from the remote procedure call and display the JList in a JFrame.

The weather service deploys in the same manner as the simple message service. Start the Tomcat servlet engine and go to the SOAP administration tool (localhost:8080/soap/admin) as shown in Fig. 29.7. Click Deploy and fill in the information in the form.
// Fig. 29.6: WeatherServiceClient.java
// WeatherServiceClient accesses the WeatherService remote
// object via SOAP to retrieve weather information.
package com.deitel.advjhtp1.soap.weather;

// Java core packages
import java.util.*;
import java.net.*;

// Java extension packages
import javax.swing.*;

// third-party packages
import org.apache.soap.*;
import org.apache.soap.rpc.*;

// Deitel packages
import com.deitel.advjhtp1.rmi.weather.*;

public class WeatherServiceClient extends JFrame {

    // WeatherServiceClient constructor
    public WeatherServiceClient( String server )
    {
        super( "SOAP WeatherService Client" );

        // connect to server and get weather information
        try {

            // URL of remote SOAP object
            URL url = new URL( "http://" + server + ":8080/soap/
            + "servlet/rpcrouter" );

            // build SOAP RPC call
            Call remoteMethod = new Call();
            remoteMethod.setTargetObjectURI( "urn:xml-weather-service" );

            // set name of remote method to be invoked
            remoteMethod.setMethodName( "getWeatherInformation" );
            remoteMethod.setEncodingStyleURI( Constants.NS_URI_SOAP_ENC );

            // invoke remote method
            Response response = remoteMethod.invoke( url, "" );

            // get response
            if ( response.generatedFault() ) {
                Fault fault = response.getFault();
            }

        } catch (MalformedURLException e) {
            System.out.println( "Error: " + e.getMessage() );
        }
    }
}
System.err.println( "CALL FAILED:\nFault Code = " + fault.getFaultCode() + "\nFault String = " + fault.getFaultString() );
}

else {
    Parameter result = response.getReturnValue();

    Vector weatherStrings = ( Vector ) result.getValue();

    // get weather information from result object
    List weatherInformation = createBeans( weatherStrings );

    // create WeatherListModel for weather information
    ListModel weatherListModel =
        new WeatherListModel( weatherInformation );

    // create JList, set its CellRenderer and add to layout
    JList weatherJList = new JList( weatherListModel );
    weatherJList.setCellRenderer( new WeatherCellRenderer() );
    getContentPane().add( new JScrollPane( weatherJList ) );
}

} // end try

// handle bad URL
catch ( MalformedURLException malformedURLException ) {
    malformedURLException.printStackTrace();
}

// handle SOAP exception
catch ( SOAPException soapException ) {
    soapException.printStackTrace();
}

} // end WeatherServiceClient constructor

// create List of WeatherBeans from Vector of Strings
public List createBeans( Vector weatherStrings ) {
    List list = new ArrayList();
    for ( int i = 0; ( weatherStrings.size() - 1 ) > i; i += 3 ) {
        list.add( new WeatherBean( ( String ) weatherStrings.elementAt( i ),
            ( String ) weatherStrings.elementAt( i + 1 ),
            ( String ) weatherStrings.elementAt( i + 2 ) ) );
    }

Fig. 29.6 SOAP implementation of class WeatherServiceClient.
Chapter 29

Introduction to Web Services and SOAP

105
106       return list;
107     }
108
109     // execute WeatherServiceClient
110    public static void main( String args[] )
111    {
112        WeatherServiceClient client = null;
113
114        // if no server IP address or host name specified,
115        // use "localhost"; otherwise use specified host
116        if ( args.length == 0 )
117            client = new WeatherServiceClient( "localhost" );
118        else
119            client = new WeatherServiceClient( args[ 0 ] );
120
121        // configure and display application window
122        clientsetDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
123        client.pack();
124        client.setResizable( false );
125        client.setVisible( true );
126    }
127 }

Fig. 29.6 SOAP implementation of class WeatherServiceClient.

as it appears in Fig. 29.8. Be sure to enter the fully qualified package name of WeatherService. Once the service is deployed, run WeatherServiceClient. The remote procedure call retrieves the weather information and, the client displays it (Fig. 29.9).

Fig. 29.7 Apache SOAP Admin page.
29.4 Internet and World Wide Web Resources


This is Sun Microsystems's site for the Open Net Environment Web services architecture.
xml.apache.org/soap
Apache’s SOAP implementation can be downloaded from this site, which also provides information and documentation.

xml.apache.org/xerces-j/index.html
Apache’s Xerces parser for Java can be downloaded from this site, along with documentation.

jakarta.apache.org
Web site for Apache’s Jakarta-Tomcat servlet engine.

**SUMMARY**

- A Web service can be any Web-accessible application, such as a Web page with dynamic content.
- A narrower definition of a Web service is an application that exposes a public interface usable by other applications over the Web.
- Sun’s ONE requires a Web service to be accessible through HTTP and other Web protocols, to communicate using XML-based messages and to be registered with a lookup service.
- Web services can provide great interoperability between diverse systems. The interoperability and scalability of Web services means developers can rapidly create large applications and larger Web services from small Web services.
- Sun’s Open Net Environment describes an architecture for creating smart Web services. According to Sun, smart Web services share a common operating environment with other services.
- SOAP is an HTTP–XML-based protocol that allows applications to communicate over the Internet, using XML documents called SOAP messages.
- SOAP is both platform and software independent and can be implemented in any programming language. SOAP supports transport using almost any conceivable protocol.
- A SOAP message contains an envelope, which describes the content, intended recipient and processing requirements of a message. The optional header element of a SOAP message specifies additional processing information for applications that receive the SOAP message.
- Through the header, more complex protocols can be built onto SOAP. The body of a SOAP message contains application-specific data for the intended recipient of the message.
- SOAP can be used to make a Remote Procedure Call (RPC), which is a request made to another machine to run a task. The RPC uses an XML vocabulary to specify the method to be invoked, any parameters the method takes and the URI of the target object.
- Because businesses use different platforms, applications and data specifications, exchanging data can be difficult. Business partners therefore establish protocols and data formats to engage in electronic commerce.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>application-to-application (A2A) integration messaging</td>
<td>messaging</td>
</tr>
<tr>
<td>asynchronous RPC</td>
<td>org.apache.soap.rpc</td>
</tr>
<tr>
<td>Call class</td>
<td>Parameter class</td>
</tr>
<tr>
<td>deploying a service</td>
<td>Remote Procedure Call (RPC)</td>
</tr>
<tr>
<td>distributed object architecture</td>
<td>request–response</td>
</tr>
<tr>
<td>Fault class</td>
<td>schema</td>
</tr>
<tr>
<td>firewall</td>
<td>setMethodName method of class Call</td>
</tr>
<tr>
<td>Hypertext Transfer Protocol (HTTP)</td>
<td>setParams method of class Call</td>
</tr>
<tr>
<td>invoke method of class Call</td>
<td>Simple Object Access Protocol (SOAP)</td>
</tr>
<tr>
<td>loosely coupled messaging</td>
<td>Sun Open Net Environment</td>
</tr>
</tbody>
</table>
synchronous RPC
Universal Description, Discovery and Integration (UDDI)

SELF-REVIEW EXERCISES

29.1 State whether each of the following is true or false. If false, explain why.
   a) SOAP is a technology for facilitating data transfer across a network.
   b) SOAP must be bound to HTTP in order to work.
   c) In order to communicate with SOAP, software systems must have the same distributed object architecture.
   d) The body of a SOAP message can contain a Remote Procedure Call.

29.2 Fill in the blanks in each of the following statements:
   a) A SOAP RPC requires the name of the method being called, its parameters and __________.
   b) A SOAP __________ contains information that describes the content, recipient and processing requirements of a SOAP message.
   c) SOAP can pass through most firewalls because __________ is its transport mechanism.
   d) SOAP RPCs use the HTTP __________ model.

ANSWERS TO SELF-REVIEW EXERCISES

29.1 a) True. b) False. SOAP can be bound to other protocols. c) False. SOAP is platform independent. d) True.

29.2 a) the processing requirements of the message. b) envelope. c) HTTP. d) request–response.

EXERCISES

29.3 Write a server-side class with a sort method that can sort given numbers. Write a client-side program that can make SOAP RPC invoke the sort method by sending a set of unsorted numbers. Display the results of sorting on the client.

29.4 Rewrite class WeatherServiceClient to update its information at a user-defined interval. Change the settings of Tomcat to make the WeatherService object persistent. This make the updates more efficient.

29.5 Write a client and server with a similar architecture to the weather service, but that obtains pricing information from a price-comparison site, such as shopper.cnet.com.

29.6 Write a server-side class that stores and retrieves Strings. Deploy the class so it is persistent on the server. Then write a client that stores and retrieves Strings from the server.

29.7 Write a simple peer-to-peer instant-messaging service. Write a server class with a method that opens a window with a text message when it is called by a client. The client allows the user to enter a text message and call the method on the server class to display the message on the other machine.

WORKS CITED

Creating Markup with XML

Objectives

• To create custom markup using XML.
• To understand the concept of an XML parser.
• To use elements and attributes to mark up data.
• To understand the difference between markup text and character data.
• To understand the concept of a well-formed XML document.
• To understand the concept of an XML namespace.
• To be able to use CDATA sections and processing instructions.

The chief merit of language is clearness, and we know that nothing detracts so much from this as do unfamiliar terms.
Galen

Every country has its own language, yet the subjects of which the untutored soul speaks are the same everywhere.
Tertullian

The historian, essentially, wants more documents than he can really use; the dramatist only wants more liberties than he can really take.
Henry James

Entities should not be multiplied unnecessarily.
William of Occam
A.1 Introduction

The Extensible Markup Language (XML) is a technology for marking up structured data so that any software with an XML parser can understand and use its content. Data independence, the separation of content from its presentation, is the essential characteristic of XML. XML documents are simply text files that are marked up in a special way, so XML is intelligible to both humans and machines. Any application can conceivably process XML data. This makes XML ideal for data exchange.

Platform independence, the separation of an application from the platform on which it runs, is the essential characteristic of Java. With Java, software developers can write a program once, and it will run on any platform that has an implementation of the Java virtual machine. Java and XML have common goals. Java allows the portability of executable code across platforms. Likewise, XML allows the portability of structured data across applications. When used together, these technologies allow applications and their associated data to work on any computer. Recognizing this fact, software developers across the world are integrating XML into their Java applications to gain Web functionality and interoperability.

In this appendix, we show how to incorporate XML into Java applications. We use a Java application we created, named ParserTest, to output an XML document’s contents. This application is included in the Appendix A examples directory on the CD-ROM that accompanies this book.

A.2 Introduction to XML Markup

In this section, we begin marking up data using XML. Consider a simple XML document (first.xml) that marks up a message (Fig. A.1). We output the entire XML document to the command line.
The document begins with the optional *XML declaration* in line 1. This declaration identifies the document as an XML document. The *version* information parameter specifies the version of XML used in the document.

**Portability Tip A.1**
*Although the XML declaration is optional, it should be used to identify the XML version to which the document conforms. Otherwise, in the future, a document without an XML declaration might be assumed to conform to the latest version of XML. Errors or other serious problems may result.*

**Common Programming Error A.1**
*Placing anything, including whitespace (i.e., spaces, tabs and newlines), before an XML declaration is an error.*

Lines 3–4 are comments, which begin with `<!--` and end with `-->`. Comments can be placed almost anywhere in an XML document and can span multiple lines. For example, we could have written lines 3–4 as

```
<!-- Fig. A.1 : first.xml
Simple introduction to XML markup -->
```

**Common Programming Error A.2**
*Placing `--` between `<!--` and `-->` is an error.*

In XML, data are marked up using *tags*, which are names enclosed in *angle brackets* (`< >`). Tags are used in pairs to delimit the beginning and end of markup. A tag that begins markup is called a *start tag* and a tag that terminates markup is called an *end tag*. Examples of start tags are `<myMessage>` and `<message>` (lines 6–7). End tags differ from start tags in that they contain a *forward slash* (`/`) character. Examples of end tags are `</message>` and `</myMessage>` in lines 7–8. XML documents can contain any number of tags.
**Good Programming Practice A.1**

XML elements and attribute names should be meaningful. For example, use `<address>` instead of `<adr>.

**Common Programming Error A.3**

Using spaces in an XML element name or attribute name is an error.

Individual units of markup (i.e., everything from a start tag to an end tag, inclusive) are called elements, which are the most fundamental building blocks of an XML document. XML documents contain exactly one element—called a root element (e.g., `myMessage` in lines 6–8)—that contains all other elements in the document. Elements are embedded or nested within each other to form hierarchies—with the root element at the top of the hierarchy. This practice allows document authors to create explicit relationships between data.

**Common Programming Error A.4**

Improperly nesting XML tags is an error. For example, `<x><y>hello</x></y>` is an error; here the nested `<y>` tag must end before the `<x>` tag.

**Good Programming Practice A.2**

When creating an XML document, add whitespace to emphasize the document’s hierarchical structure. This makes documents more readable to humans.

**Common Programming Error A.5**

Attempting to create more than one root element in an XML document is an error.

Elements, such as the root element, that contain other elements are called parent elements. Elements nested within a parent element are called children. Parent elements can have any number of children, but an individual child element can have only one parent. As we will see momentarily, it is possible for an element to be both a parent element and a child element. Element `message` is an example of a child element and element `myMessage` is an example of a parent element.

**Common Programming Error A.6**

XML element names are case sensitive. Using the wrong mixture of case is an error. For example, using the start tag `<message>` and end tag `</Message>` is an error.

In addition to being placed between tags, data can be placed in attributes, which are name-value pairs in start tags. Elements can have any number of attributes. In Fig. A.1, attribute `id` is assigned the value "643070". XML element and attribute names can be of any length and may contain letters, digits, underscores, hyphens and periods; they must begin with a letter or an underscore.

**Common Programming Error A.7**

Not placing an attribute’s value in either single or double quotes is a syntax error.

Notice that the XML declaration output differs from the XML declaration in line 1. The optional `encoding` declaration specifies the method used to represent characters electronically. **UTF-8** is a character encoding typically used for Latin-alphabet characters.
(e.g., English) that can be stored in one byte. When present, this declaration allows authors to specify a character encoding explicitly. When omitted, either UTF-8 or UTF-16 (a format for encoding and storing characters in two bytes) is the default. We discuss character encoding in Section A.4.

**Portability Tip A.2**

The encoding declaration allows XML documents to be authored in a wide variety of human languages.

### A.3 Parsers and Well-Formed XML Documents

A software program called an XML parser (or an XML processor) is required to process an XML document. XML parsers read the XML document, check its syntax, report any errors and allow programmatic access to the document’s contents. An XML document is considered well formed if it is syntactically correct (i.e., errors are not reported by the parser when the document is processed). Figure A.1 is an example of a well-formed XML document.

If an XML document is not well formed, the parser reports errors. For example, if the end tag (line 8) in Fig. A.1 is omitted, the error message shown in Fig. A.2 is generated by the parser.

```xml
<?xml version = "1.0"?>
<!-- Fig. A.2 : error.xml -->
<!-- XML document missing an end tag -->
<myMessage id = "643070">
<message>Welcome to XML!</message>
</myMessage>
```

```
C:\>java -jar ParserTest.jar error.xml
Exception in thread "main" org.xml.sax.SAXParseException: End of entity not allowed; an end tag is missing.
  at org.apache.crimson.parser.Parser2.fatal(Parser2.java:3035)
  at org.apache.crimson.parser.Parser2.fatal(Parser2.java:3023)
  at org.apache.crimson.parser.Parser2.content(Parser2.java:1758)
  at org.apache.crimson.parser.Parser2.maybeElement(Parser2.java:1468)
  at org.apache.crimson.parser.Parser2.parseInternal(Parser2.java:499)
  at org.apache.crimson.parser.Parser2.parse(Parser2.java:304)
  at org.apache.crimson.parser.XMLReaderImpl.parse(XMLReaderImpl.java:433)
  at javax.xml.parsers.DocumentBuilder.parse(DocumentBuilder.java:161)
  at ParserTest.main(ParserTest.java:42)
```

**Fig. A.2** XML document missing an end tag.
Most XML parsers can be downloaded at no charge. Several Independent Software Vendors (ISVs) have developed XML parsers, which can be found at [www.oasis-open.org/cover/xml.html#xmlparsers](http://www.oasis-open.org/cover/xml.html#xmlparsers). In this appendix, we will use the reference implementation for the Java API for XML Processing 1.1 (JAXP).

Parsers can support the Document Object Model (DOM) and/or the Simple API for XML (SAX) for accessing a document’s content programmatically, using languages such as Java™, Python, and C. A DOM-based parser builds a tree structure containing the XML document’s data in memory. A SAX-based parser processes the document and generates events (i.e., notifications to the application) when tags, text, comments, etc., are encountered. These events return data from the XML document. Software programs can “listen” for the events to obtain data from the XML document.

The examples we present use DOM-based parsing. In Appendix C, we provide a detailed discussion of the DOM. We do not discuss SAX-based parsing in these appendices.

### A.4 Characters

In this section, we discuss the collection of characters—called a character set—permitted in an XML document. XML documents may contain: carriage returns, line feeds and Unicode® characters. Unicode is a character set created by the Unicode Consortium ([www.unicode.org](http://www.unicode.org)), which encodes the vast majority of the world’s commercially viable languages. We discuss Unicode in detail in Appendix I.

#### A.4.1 Characters vs. Markup

Once a parser determines that all characters in a document are legal, it must differentiate between markup text and character data. Markup text is enclosed in angle brackets ( and ). Character data (sometimes called element content) is the text delimited by the start tag and end tag. Child elements are considered markup—not character data. Lines 1, 3–4 and 6–8 in Fig. A.1 contain markup text. In line 7, the tags `<message>` and `</message>` are the markup text and the text **Welcome to XML!** is character data.

#### A.4.2 White Space, Entity References and Built-In Entities

Spaces, tabs, line feeds and carriage returns are characters commonly called whitespace characters. An XML parser is required to pass all characters in a document, including whitespace characters, to the application (e.g., a Java application) using the XML document.

Figure A.3 demonstrates that whitespace characters are passed by the parser to the application using the XML document. In this case, we simply print the data returned by the parser.

```xml
1  <?xml version="1.0"?>
2
3  <!-- Fig. A.3 : whitespace.xml -->
4  <!-- Demonstrating whitespace, entities -->
5  <!-- and empty elements -->
6
Fig. A.3  Whitespace characters in an XML document (part 1 of 2).
A parser can inform an application as to whether individual whitespace characters are significant (i.e., need to be preserved) or insignificant (i.e., need not be preserved). The output window illustrates that the majority of whitespace characters in the document are considered significant. Line 2 was considered insignificant by the application as well as the extra space characters in the start tag of line 13. In Appendix B, you will see that whitespace may or may not be significant, depending on the Document Type Definition (DTD) that an XML file uses. We will explore the subtleties of whitespace interpretation in greater detail in Appendix B.

XML element markup consists of a start tag, character data and an end tag. The element of line 10 is called an empty element, because it does not contain character data between its start and end tags. The forward slash character closes the tag. Alternatively, this empty element can be written as

```xml
<company name = "Deitel & Associates, Inc." />
```

Both forms are equivalent.

Almost any character can be used in an XML document, but the characters `&` and `<>` are reserved in XML and may not be used in character data. To use these symbols in character data or in attribute values, entity references must be used. Entity references begin with an ampersand (`&`) and end with a semicolon (`;`). XML provides entity references (or built-in entities) for the ampersand (`&amp;`), left-angle bracket (`&lt;`), right angle bracket (`&gt;`), apostrophe (`&apos;`) and quotation mark (`&quot;`).
**Common Programming Error A.8**

Attempting to use the left-angle bracket (<) in character data or in attribute values is an error.

**Common Programming Error A.9**

Attempting to use the ampersand (&)—other than in an entity reference—in character data or in attribute values is an error.

### A.5 CDATA Sections and Processing Instructions

In this section, we discuss parts of an XML document, called *CDATA sections*, that can contain text, reserved characters (e.g., `<`) and whitespace characters. Character data in a CDATA section are not processed by the XML parser. A common use of a CDATA section is for programming code such as JavaScript and C++, which often include the characters `&` and `<. Figure A.4 presents an XML document that compares text in a CDATA section with character data.

The first *sample* element (lines 8–12) contains C++ code as character data. Each occurrence of `<`, `>`, and `&` is replaced by an entity reference. Lines 15–20 use a CDATA section to indicate a block of text that the parser should not treat as character data or markup. CDATA sections begin with `<![CDATA[` and terminate with `]]>`. Notice that the `<` and `&` characters (lines 18–19) do not need to be replaced by entity references.

```xml
<?xml version = "1.0"?>
<!-- Fig. A.4 : cdata.xml -->
<!-- CDATA section containing C++ code -->

<book title = "C++ How to Program" edition = "3">
   <sample>
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
         cerr &lt;&lt; this-&gt;displayError();
   </sample>

   <sample>
   <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
         cerr &lt;&lt; this-&gt;displayError();
   ]]]>
   </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>
```

**Fig. A.4** Using a CDATA section (part 1 of 2).
Appendix A Creating Markup with XML 1619

Common Programming Error A.10

Placing one or more spaces inside the opening <! [CDATA [ or closing ]] > is an error.

Because a CDATA section is not parsed, it can contain almost any text, including characters normally reserved for XML syntax, such as < and &. However, CDATA sections cannot contain the text ]]> , because this is used to terminate a CDATA section. For example,

<!-- Fig. A.4 : cdata.xml -->
<book title="C++ How to Program" edition="3">
  <sample>
    // C++ comment
    if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
      cerr &lt;&lt; this-&gt;displayError();
  </sample>

  <sample>
    <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
        cerr &lt;&lt; this-&gt;displayError();
    ]]>]
  </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>

Fig. A.4 Using a CDATA section (part 2 of 2).

C:\>java -jar ParserTest.jar cdata.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. A.4 : cdata.xml -->
<book title="C++ How to Program" edition="3">
  <sample>
    // C++ comment
    if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
      cerr &lt;&lt; this-&gt;displayError();
  </sample>

  <sample>
    <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
        cerr &lt;&lt; this-&gt;displayError();
    ]]>]
  </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>

Common Programming Error A.10

Placing one or more spaces inside the opening <! [CDATA [ or closing ]] > is an error.

Because a CDATA section is not parsed, it can contain almost any text, including characters normally reserved for XML syntax, such as < and &. However, CDATA sections cannot contain the text ]]> , because this is used to terminate a CDATA section. For example,

<!-- Fig. A.4 : cdata.xml -->
<book title="C++ How to Program" edition="3">
  <sample>
    // C++ comment
    if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
      cerr &lt;&lt; this-&gt;displayError();
  </sample>

  <sample>
    <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
        cerr &lt;&lt; this-&gt;displayError();
    ]]>]
  </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>

Fig. A.4 Using a CDATA section (part 2 of 2).

Common Programming Error A.10

Placing one or more spaces inside the opening <! [CDATA [ or closing ]] > is an error.

Because a CDATA section is not parsed, it can contain almost any text, including characters normally reserved for XML syntax, such as < and &. However, CDATA sections cannot contain the text ]]> , because this is used to terminate a CDATA section. For example,

<!-- Fig. A.4 : cdata.xml -->
<book title="C++ How to Program" edition="3">
  <sample>
    // C++ comment
    if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
      cerr &lt;&lt; this-&gt;displayError();
  </sample>

  <sample>
    <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
        cerr &lt;&lt; this-&gt;displayError();
    ]]>]
  </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>

Fig. A.4 Using a CDATA section (part 2 of 2).

Common Programming Error A.10

Placing one or more spaces inside the opening <! [CDATA [ or closing ]] > is an error.

Because a CDATA section is not parsed, it can contain almost any text, including characters normally reserved for XML syntax, such as < and &. However, CDATA sections cannot contain the text ]]> , because this is used to terminate a CDATA section. For example,

<!-- Fig. A.4 : cdata.xml -->
<book title="C++ How to Program" edition="3">
  <sample>
    // C++ comment
    if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
      cerr &lt;&lt; this-&gt;displayError();
  </sample>

  <sample>
    <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
        cerr &lt;&lt; this-&gt;displayError();
    ]]>]
  </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>

Fig. A.4 Using a CDATA section (part 2 of 2).

Common Programming Error A.10

Placing one or more spaces inside the opening <! [CDATA [ or closing ]] > is an error.

Because a CDATA section is not parsed, it can contain almost any text, including characters normally reserved for XML syntax, such as < and &. However, CDATA sections cannot contain the text ]]> , because this is used to terminate a CDATA section. For example,

<!-- Fig. A.4 : cdata.xml -->
<book title="C++ How to Program" edition="3">
  <sample>
    // C++ comment
    if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
      cerr &lt;&lt; this-&gt;displayError();
  </sample>

  <sample>
    <![CDATA[
      // C++ comment
      if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
        cerr &lt;&lt; this-&gt;displayError();
    ]]>]
  </sample>

C++ How to Program by Deitel &amp; Deitel

<?button cpp = "sample.cpp" ansi = "yes"?>
</book>
Software Engineering Observation A.1

Processing instructions provide a means for programmers to insert application-specific information into an XML document without affecting the document’s portability.

A.6 XML Namespaces

Because XML allows document authors to create their own tags, naming collisions (i.e., conflicts between two different elements that have the same name) can occur. For example, we may use the element book to mark up data about one of our publications. A stamp collector may also create an element book to mark up data about a book of stamps. If both of these elements were used in the same document, there would be a naming collision, and it would be difficult to determine which kind of data each element contained. In this section, we discuss a method for preventing collisions called namespaces. In Appendix D, we begin using namespaces.

For example,

```xml
<subject>Math</subject>
```

and

```xml
<subject>Thrombosis</subject>
```

use a subject element to mark up a piece of data. However, in the first case the subject is something one studies in school, whereas in the second case the subject is in the field of medicine. These two subject elements can be differentiated using namespaces. For example

```xml
<school:subject>Math</school:subject>
```

and

```xml
<medical:subject>Thrombosis</medical:subject>
```

indicate two distinct subject elements. Both school and medical are namespace prefixes. Namespace prefixes are prepended to element and attribute names in order to specify the namespace in which the element or attribute can be found. Each namespace prefix is tied to a uniform resource identifier (URI) that uniquely identifies the namespace. Document authors can create their own namespace prefixes, as shown in Fig. A.5 (lines 6–7). Virtually any name may be used for a namespace, except the reserved namespace xml.

```
<?xml version = "1.0"?>
<!-- Fig. A.5 : namespace.xml -->
<!-- Namespaces -->
<text:directory xmlns:text = "urn:deitel:textInfo"
xmlns:image = "urn:deitel:imageInfo">
```

Fig. A.5  Demonstrating XML namespaces (part 1 of 2).
In Fig. A.5, two distinct file elements are differentiated using namespaces. Lines 6–7 use the XML namespace keyword xmlns to create two namespace prefixes: text and image. The values assigned to attributes xmlns:text and xmlns:image are called Uniform Resource Identifiers (URIs). By definition, a URI is a series of characters used to differentiate names.

To ensure that a namespace is unique, the document author must provide a unique URI. Here, we use the text urn:deitel:textInfo and urn:deitel:imageInfo as URIs. A common practice is to use Universal Resource Locators (URLs) for URIs, because the domain names (e.g., deitel.com) used in URLs are guaranteed to be unique. For example, lines 6–7 could have been written as

```xml
  <text:file filename="book.xml">
    <text:description>A book list</text:description>
  </text:file>
  <image:file filename="funny.jpg">
    <image:description>A funny picture</image:description>
    <image:size width="200" height="100"/>
  </image:file>
</text:directory>
```

where we use URLs related to the Deitel & Associates, Inc., domain name (www.deitel.com). These URLs are never visited by the parser—they only represent a series of characters for differentiating names and nothing more. The URLs need not even exist or be properly formed.

Lines 9–11 use the namespace prefix text to describe elements file and description. Notice that end tags have the namespace prefix text applied to them as well. Lines 13–16 apply namespace prefix image to elements file, description and size.

Fig. A.5 Demonstrating XML namespaces (part 2 of 2).

In Fig. A.5, two distinct file elements are differentiated using namespaces. Lines 6–7 use the XML namespace keyword xmlns to create two namespace prefixes: text and image. The values assigned to attributes xmlns:text and xmlns:image are called Uniform Resource Identifiers (URIs). By definition, a URI is a series of characters used to differentiate names.

To ensure that a namespace is unique, the document author must provide a unique URI. Here, we use the text urn:deitel:textInfo and urn:deitel:imageInfo as URIs. A common practice is to use Universal Resource Locators (URLs) for URIs, because the domain names (e.g., deitel.com) used in URLs are guaranteed to be unique. For example, lines 6–7 could have been written as

```xml
  <text:file filename="book.xml">
    <text:description>A book list</text:description>
  </text:file>
  <image:file filename="funny.jpg">
    <image:description>A funny picture</image:description>
    <image:size width="200" height="100"/>
  </image:file>
</text:directory>
```

where we use URLs related to the Deitel & Associates, Inc., domain name (www.deitel.com). These URLs are never visited by the parser—they only represent a series of characters for differentiating names and nothing more. The URLs need not even exist or be properly formed.

Lines 9–11 use the namespace prefix text to describe elements file and description. Notice that end tags have the namespace prefix text applied to them as well. Lines 13–16 apply namespace prefix image to elements file, description and size.
To eliminate the need to place a namespace prefix in each element, authors may specify a *default namespace* for an element and all of its child elements. Figure A.6 demonstrates the use of default namespaces.

We declare a default namespace using the `xmlns` attribute with a URI as its value (line 6). Once this default namespace is in place, child elements that are part of the namespace do not need a namespace prefix. Element `file` (line 9) is in the namespace corresponding to the URI `urn:deitel:textInfo`. Compare this usage with that in Fig. A.5, where we prefixed the `file` and `description` elements with the namespace prefix `text` (lines 9–11).

The default namespace applies to all elements contained in the `directory` element. However, we may use a namespace prefix to specify a different namespace for particular
A.7 Internet and World Wide Web Resources

**www.w3.org/XML**
Worldwide Web Consortium Extensible Markup Language home page. Contains links to related XML technologies, recommended books, a time-line for publications, developer discussions, translations, software, etc.

**www.w3.org/Addressing**
Worldwide Web Consortium addressing home page. Contains information on URIs and links to other resources.

**www.xml.com**
This is one of the most popular XML sites on the Web. It has resources and links relating to all aspects of XML, including articles, news, seminar information, tools, Frequently Asked Questions (FAQs), etc.

**www.xml.org**
“The XML Industry Portal” is another popular XML site that includes links to many different XML resources, such as news, FAQs and descriptions of XML-derived markup languages.

**www.oasis-open.org/cover**
Oasis XML Cover Pages home page is a comprehensive reference for many aspects of XML and its related technologies. The site includes links to news, articles, software and events.

**html.about.com/compute/html/cs/xmlandjava/index.htm**
This site contains articles about XML and Java and is updated regularly.

**www.w3schools.com/xml**
Contains a tutorial that introduces the reader to the major aspects of XML. The tutorial contains many examples.

**java.sun.com/xml**
Home page of the Sun’s JAXP and parser technology.

**SUMMARY**
- XML is a technology for creating markup languages to describe data of virtually any type in a structured manner.
- XML allows document authors to describe data precisely by creating their own tags. Markup languages can be created using XML for describing almost anything.
- XML documents are commonly stored in text files that end in the extension .xml. Any text editor can be used to create an XML document. Many software packages allow data to be saved as XML documents.
- The XML declaration specifies the version to which the document conforms.
- All XML documents must have exactly one root element that contains all of the other elements.
- To process an XML document, a software program called an XML parser is required. The XML parser reads the XML document, checks its syntax, reports any errors and allows access to the document’s contents.
- An XML document is considered well formed if it is syntactically correct (i.e., the parser did not report any errors due to missing tags, overlapping tags, etc.). Every XML document must be well formed.
• Parsers may or may not support the Document Object Model (DOM) and/or the Simple API for XML (SAX) for accessing a document’s content programmatically by using languages such as Java, Python and C.

• XML documents may contain: carriage return, the line feed and Unicode characters. Unicode is a standard that was released by the Unicode Consortium in 1991 to expand character representation for most of the world’s major languages. The American Standard Code for Information Interchange (ASCII) is a subset of Unicode.

• Markup text is enclosed in angle brackets (i.e., < and >). Character data are the text between a start tag and an end tag. Child elements are considered markup—not character data.

• Spaces, tabs, line feeds and carriage returns are whitespace characters. In an XML document, the parser considers whitespace characters to be either significant (i.e., preserved by the parser) or insignificant (i.e., not preserved by the parser).

• Almost any character may be used in an XML document. However, the characters ampersand (&) and left-angle bracket (<) are reserved in XML and may not be used in character data, except in CDATA sections. Angle brackets are reserved for delimiting markup tags. The ampersand is reserved for delimiting hexadecimal values that refer to a specific Unicode character. These expressions are terminated with a semicolon (;) and are called entity references. The apostrophe and double-quote characters are reserved for delimiting attribute values.

• XML provides built-in entities for ampersand (&amp;), left-angle bracket (&lt;), right-angle bracket (&gt;), apostrophe (&apos;) and quotation mark (&quot;).

• All XML start tags must have a corresponding end tag and all start- and end tags must be properly nested. XML is case sensitive, therefore start tags and end tags must have matching capitalization.

• Elements define a structure. An element may or may not contain content (i.e., child elements or character data). Attributes describe elements. An element may have zero, one or more attributes associated with it. Attributes are nested within the element’s start tag. Attribute values are enclosed in quotes—either single or double.

• XML element and attribute names can be of any length and may contain letters, digits, underscores, hyphens and periods; and they must begin with either a letter or an underscore.

• A processing instruction’s (PI’s) information is passed by the parser to the application using the XML document. Document authors may create their own processing instructions. Almost any name may be used for a PI target except the reserved word xml (in any mixture of case). Processing instructions allow document authors to embed application-specific data within an XML document. This data are not intended to be readable by humans, but readable by applications.

• CDATA sections may contain text, reserved characters (e.g., <), words and whitespace characters. XML parsers do not process the text in CDATA sections. CDATA sections allow the document author to include data that is not intended to be parsed. CDATA sections cannot contain the text ]]>.

• Because document authors can create their own tags, naming collisions (e.g., conflicts that arise when document authors use the same names for elements) can occur. Namespaces provide a means for document authors to prevent naming collisions. Document authors create their own namespaces. Virtually any name may be used for a namespace, except the reserved namespace xml.

• A Universal Resource Identifier (URI) is a series of characters used to differentiate names. URIs are used with namespaces.

**TERMINOLOGY**

- `<![CDATA[ and ]]>` to delimit a CDATA section
- `&` to delimit an ampersand (&amp;)
- `>` to delimit an angle brackets (< and >)
- `apos;` to delimit an apostrophe (&apos;)
- `quot;` to delimit a quotation mark (&quot;)
- `xml` to delimit a processing instruction
Appendix A Creating Markup with XML

SELF-REVIEW EXERCISES

A.1 State whether the following are true or false. If false, explain why.
   a) XML is a technology for creating markup languages.
   b) XML markup text is delimited by forward and backward slashes (/ and \).
   c) All XML start tags must have corresponding end tags.
   d) Parsers check an XML document’s syntax and may support the Document Object Model and/or the Simple API for XML.
   e) An XML document is considered well formed if it contains whitespace characters.
   f) SAX-based parsers process XML documents and generate events when tags, text, comments, etc., are encountered.
   g) When creating new XML tags, document authors must use the set of XML tags provided by the W3C.
   h) The pound character (#), the dollar sign ($), the ampersand (&), the greater-than symbol (>) and the less-than symbol (<) are examples of XML reserved characters.
   i) Any text file is automatically considered to be an XML document by a parser.

A.2 Fill in the blanks in each of the following statements:
   a) A/An ________ processes an XML document.
   b) Valid characters that can be used in an XML document are the carriage return, line feed and ________ characters.
   c) An entity reference must be proceeded by a/an ________ character.
   d) A/An ________ is delimited by <? and ?>.
   e) Text in a/an ________ section is not parsed.
f) An XML document is considered ________ if it is syntactically correct.
g) _________ help document authors prevent element-naming collisions.
h) A/An _________ tag does not contain character data.
i) The built-entity for the ampersand is _________.

A.3 Identify and correct the error(s) in each of the following:
a) `<my Tag>This is my custom markup</my Tag>`
b) `<!PI value!> <!-- a sample processing instruction -->`
c) `<myXML>I know XML!!</MyXML>`
d) `<CDATA>This is a CDATA section.</CDATA>`
e) `<xml>x < 5 && x > y</xml> <!-- mark up a Java condition -->`

ANSWERS TO SELF-REVIEW EXERCISES

A.4 a) True. b) False. In an XML document, markup text is any text delimited by angle brackets (`<` and `>`) with a forward slash being used in the end tag. c) True. d) True. e) False. An XML document is considered well formed if it is parsed successfully. f) True. g) False. When creating new tags, programmers may use any valid name except the reserved word `xml` (in any mixture of case). h) False. XML reserved characters include the ampersand (`&`) and the left angle bracket (`<`), but not the right-angle bracket (`>`), `#`, and `$`. i) False. The text file must be parsable by an XML parser. If parsing fails, the document cannot be considered an XML document.

A.5 a) parser. b) Unicode. c) ampersand (`&`). d) processing instruction. e) CDATA. f) well formed. g) namespaces. h) empty. i) `&amp;`.

A.6 a) Element name `my tag` contains a space. The forward slash, `/`, is missing in the end tag. The corrected markup is `<myTag>This is my custom markup</myTag>`
b) Incorrect delimiters for a processing instruction. The corrected markup is `<?PI value?> <!-- a sample processing instruction -->`
c) Incorrect mixture of case in end tag. The corrected markup is `<myXML>I know XML!!</MyXML>` or `<MyXML>I know XML!!</MyXML>`
d) Incorrect syntax for a CDATA section. The corrected markup is `<![CDATA[This is a CDATA section.]]>`
e) The name `xml` is reserved and cannot be used as an element. The characters `<`, `&` and `>` must be represented using entities. The closing comment delimiter should be two hyphens—not two stars. Corrected markup is `<someName>x &lt; 5 &amp;&amp; x &gt; y</someName>`
`<!-- mark up a Java condition -->`
Objectives

- To understand what a DTD is.
- To be able to write DTDs.
- To be able to declare elements and attributes in a DTD.
- To understand the difference between general entities and parameter entities.
- To be able to use conditional sections with entities.
- To be able to use NOTATIONs.
- To understand how an XML document’s whitespace is processed.

*To whom nothing is given, of him can nothing be required.*
Henry Fielding

*Like everything metaphysical, the harmony between thought and reality is to be found in the grammar of the language.*
Ludwig Wittgenstein

*Grammar, which knows how to control even kings.*
Molière
B.1 Introduction

In this appendix, we discuss Document Type Definitions (DTDs), which define an XML document’s structure (e.g., what elements, attributes, etc. are permitted in the document). An XML document is not required to have a corresponding DTD. However, DTDs are often recommended to ensure document conformity, especially in business-to-business (B2B) transactions, where XML documents are exchanged. DTDs specify an XML document’s structure and are themselves defined using EBNF (Extended Backus-Naur Form) grammar—not the XML syntax introduced in Appendix A.

B.2 Parsers, Well-Formed and Valid XML Documents

Parsers are generally classified as validating or nonvalidating. A validating parser is able to read a DTD and determine whether the XML document conforms to it. If the document conforms to the DTD, it is referred to as valid. If the document fails to conform to the DTD but is syntactically correct, it is well formed, but not valid. By definition, a valid document is well formed.

A nonvalidating parser is able to read the DTD, but cannot check the document against the DTD for conformity. If the document is syntactically correct, it is well formed.

In this appendix, we use a Java program we created to check a document conformance. This program, named Validator.jar, is located in the Appendix B examples directory. Validator.jar uses the reference implementation for the Java API for XML Processing 1.1, which requires crimson.jar and jaxp.jar.
B.3 Document Type Declaration

DTDs are introduced into XML documents using the document type declaration (i.e., `DOCTYPE`). A document type declaration is placed in the XML document’s prolog (i.e., all lines preceding the root element), begins with `<!DOCTYPE` and ends with `>`. The document type declaration can point to declarations that are outside the XML document (called the external subset) or can contain the declaration inside the document (called the internal subset). For example, an internal subset might look like

```xml
<!DOCTYPE myMessage [
  <!ELEMENT myMessage ( #PCDATA )>
]>
```

The first `myMessage` is the name of the document type declaration. Anything inside the square brackets (`[]`) constitutes the internal subset. As we will see momentarily, `ELEMENT` and `#PCDATA` are used in “element declarations.”

External subsets physically exist in a different file that typically ends with the `.dtd` extension, although this file extension is not required. External subsets are specified using either keyword the keyword `SYSTEM` or the keyword `PUBLIC`. For example, the `DOCTYPE` external subset might look like

```xml
<!DOCTYPE myMessage SYSTEM "myDTD.dtd">
```

which points to the `myDTD.dtd` document. The `PUBLIC` keyword indicates that the DTD is widely used (e.g., the DTD for HTML documents). The DTD may be made available in well-known locations for more efficient downloading. We used such a DTD in Chapters 9 and 10 when we created XHTML documents. The `DOCTYPE` uses the `PUBLIC` keyword to reference the well-known DTD for XHTML version 1.0.

XML parsers that do not have a local copy of the DTD may use the URL provided to download the DTD to perform validation.

Both the internal and external subset may be specified at the same time. For example, the `DOCTYPE`

```xml
<!DOCTYPE myMessage SYSTEM "myDTD.dtd" [
  <!ELEMENT myElement ( #PCDATA )>
]>
```

contains declarations from the `myDTD.dtd` document, as well as an internal declaration.

---

Software Engineering Observation B.1

The document type declaration’s internal subset plus its external subset form the DTD.

---

Software Engineering Observation B.2

The internal subset is visible only within the document in which it resides. Other external documents cannot be validated against it. DTDs that are used by many documents should be placed in the external subset.
### B.4 Element Type Declarations

Elements are the primary building blocks used in XML documents and are declared in a DTD with **element type declarations** (*ELEMENTs*). For example, to declare element *myMessage*, we might write

```
<!ELEMENT myElement ( #PCDATA )>
```

The element name (e.g., *myElement*) that follows **ELEMENT** is often called a *generic identifier*. The set of parentheses that follow the element name specify the element’s allowed content and is called the **content specification**. Keyword **PCDATA** specifies that the element must contain *parsable character data*. These data will be parsed by the XML parser, therefore any markup text (i.e., `<`, `>`, `&`, etc.) will be treated as markup. We will discuss the content specification in detail momentarily.

---

**Common Programming Error B.1**

**Attempting to use the same element name in multiple element type declarations is an error.**

---

Figure B.1 lists an XML document that contains a reference to an external DTD in the **DOCTYPE**. We use **Validator.jar** to check the document’s conformity against its DTD.

The document type declaration (line 6) specifies the name of the root element as **MyMessage**. The element **myMessage** (lines 8–10) contains a single child element named **message** (line 9).

Line 3 of the DTD (Fig. B.2) declares element **myMessage**. Notice that the content specification contains the name **message**. This indicates that element **myMessage** contains exactly one child element named **message**. Because **myMessage** can have only an element as its content, it is said to have **element content**. Line 4, declares element **message** whose content is of type **PCDATA**.

---

**Common Programming Error B.2**

**Having a root element name other than the name specified in the document type declaration is an error.**

---

If an XML document’s structure is inconsistent with its corresponding DTD, but is syntactically correct, the document is only well formed—not valid. Figure B.3 shows the messages generated when the required **message** element is omitted.

---

```xml
1  <?xml version = "1.0"?>
2  
3  <!-- Fig. B.1: welcome.xml -->
4  <!-- Using an external subset -->
5  
6  <!DOCTYPE myMessage SYSTEM "welcome.dtd">
7  
8  <myMessage>
9  <message>Welcome to XML!</message>
10  </myMessage>
```

---

**Fig. B.1** XML document declaring its associated DTD.
Appendix B

Document Type Definition (DTD) 1631

B.4.1 Sequences, Pipe Characters and Occurrence Indicators

DTDs allow the document author to define the order and frequency of child elements. The *comma (,)*—called a *sequence*—specifies the order in which the elements must occur. For example,

```xml
<!ELEMENT classroom ( teacher, student )>
```

specifies that element *classroom* must contain exactly one *teacher* element followed by exactly one *student* element. The content specification can contain any number of items in sequence.

Similarly, choices are specified using the *pipe character (|)*, as in

```xml
<!ELEMENT dessert ( iceCream | pastry )>
```

which specifies that element *dessert* must contain either one *iceCream* element or one *pastry* element, but not both. The content specification may contain any number of pipe character-separated choices.

An element’s frequency (i.e., number of occurrences) is specified by using either the *plus sign (+)*, *asterisk (*)* or *question mark (?)* occurrence indicator (Fig. B.4).

```xml
<?xml version = "1.0"?>
<!DOCTYPE myMessage SYSTEM "welcome.dtd">
<!-- Fig. B.3 : welcome-invalid.xml -->
<!-- well-formed, but invalid document -->
<!ELEMENT myMessage ( message )>
<!ELEMENT message ( #PCDATA )>
<myMessage>
</myMessage>
```

C:\>java -jar Validator.jar welcome-invalid.xml
error: Element "myMessage" requires additional elements.

Fig. B.3 Invalid XML document.
A plus sign indicates one or more occurrences. For example,

```xml
<!ELEMENT album ( song+ )>
```
specifies that element `album` contains one or more `song` elements.

The frequency of an element group (i.e., two or more elements that occur in some combination) is specified by enclosing the element names inside the content specification with parentheses, followed by either the plus sign, asterisk or question mark. For example,

```xml
<!ELEMENT album ( title, ( songTitle, duration )+ )>
```
indicates that element `album` contains one `title` element followed by any number of `songTitle/duration` element groups. At least one `songTitle/duration` group must follow `title`, and in each of these element groups, the `songTitle` must precede the `duration`. An example of markup that conforms to this is

```xml
<album>
  <title>XML Classical Hits</title>
  <songTitle>XML Overture</songTitle>
  <duration>10</duration>
  <songTitle>XML Symphony 1.0</songTitle>
  <duration>54</duration>
</album>
```
which contains one `title` element followed by two `songTitle/duration` groups. The asterisk (*) character indicates an optional element that, if used, can occur any number of times. For example,

```xml
<!ELEMENT library ( book* )>
```
indicates that element `library` contains any number of `book` elements, including the possibility of none at all. Markup examples that conform to this mark up are

```xml
<library>
  <book>The Iliad</book>
  <book>The Jungle</book>
</library>
```
and

```xml
<library/>
```

Optional elements that, if used, may occur only once are followed by a question mark (?). For example,

```xml
<!ELEMENT seat ( person? )>
```

indicates that element `seat` contains at most one `person` element. Examples of markup that conform to this are

```xml
<seat>
  <person>Jane Doe</person>
</seat>
```

and

```xml
<seat/>
```

Now we consider three more element type declarations and provide a declaration for each. The declaration

```xml
<!ELEMENT class ( number, ( instructor | assistant+ ),
                   ( credit | noCredit ) )>
```

specifies that a `class` element must contain a `number` element, either one `instructor` element or any number of `assistant` elements and either one `credit` element or one `noCredit` element. Markup examples that conform to this are

```xml
<class>
  <number>123</number>
  <instructor>Dr. Harvey Deitel</instructor>
  <credit>4</credit>
</class>
```

and

```xml
<class>
  <number>456</number>
  <assistant>Tem Nieto</assistant>
  <assistant>Paul Deitel</assistant>
  <credit>3</credit>
</class>
```

The declaration

```xml
<!ELEMENT donutBox ( jelly?, lemon*,
                   ( ( creme | sugar )+ | glazed ) )>
```

specifies that element `donutBox` can have zero or one `jelly` elements, followed by zero or more `lemon` elements, followed by one or more `creme` or `sugar` elements or exactly one `glazed` element. Markup examples that conform to this are
The declaration

```xml
<!ELEMENT farm ( farmer+, ( dog* | cat? ), pig*,
( goat | cow )?, ( chicken+ | duck* ) )>
```

indicates that element `farm` can have one or more `farmer` elements, any number of optional `dog` elements or an optional `cat` element, any number of optional `pig` elements, an optional `goat` or `cow` element and one or more `chicken` elements or any number of optional `duck` elements. Examples of markup that conform to this are

```xml
<farm>
  <farmer>Jane Doe</farmer>
  <farmer>John Doe</farmer>
  <cat>Lucy</cat>
  <pig>Bo</pig>
  <chicken>Jill</chicken>
</farm>
```

and

```xml
<farm>
  <farmer>Red Green</farmer>
  <duck>Billy</duck>
  <duck>Sue</duck>
</farm>
```

### B.4.2 EMPTY, Mixed Content and ANY

Elements must be further refined by specifying the types of content they contain. In the previous section, we introduced element content, indicating that an element can contain one or more child elements as its content. In this section, we introduce content specification types for describing nonelement content.

In addition to element content, three other types of content exist: `EMPTY`, `mixed content` and `ANY`. Keyword `EMPTY` declares empty elements, which do not contain character data or child elements. For example,

```xml
<!ELEMENT oven EMPTY>
```
Appendix B  Document Type Definition (DTD)  1635

declares element oven to be an empty element. The markup for an oven element would appear as

    <oven/>

or

    <oven></oven>

in an XML document conforming to this declaration.

An element can also be declared as having mixed content. Such elements may contain any combination of elements and PCDATA. For example, the declaration

    <!ELEMENT myMessage ( #PCDATA | message )*>

indicates that element myMessage contains mixed content. Markup conforming to this declaration might look like

    <myMessage>Here is some text, some
        <message>other text</message> and
        <message>even more text</message>.</myMessage>

Element myMessage contains two message elements and three instances of character data. Because of the *, element myMessage could have contained nothing.

Figure B.5 specifies the DTD as an internal subset (lines 6–10). In the prolog (line 1), we use the standalone attribute with a value of yes. An XML document is standalone if it does not reference an external subset. This DTD defines three elements: one that contains mixed content and two that contain parsed character data.

```xml
<?xml version = "1.0" standalone = "yes"?>
<!DOCTYPE format [  
   <!ELEMENT format ( #PCDATA | bold | italic )*>  
   <!ELEMENT bold ( #PCDATA )>  
   <!ELEMENT italic ( #PCDATA )>  
]>
<format>
  Book catalog entry:
  <bold>XML</bold>
  <italic>XML How to Program</italic>
  This book carefully explains XML-based systems development.
</format>
```

C:\>java -jar Validator.jar mixed.xml
Document is valid.

Fig. B.5  Example of a mixed-content element.
Line 7 declares element `format` as a mixed content element. According to the declaration, the `format` element may contain either parsed character data (`PCDATA`), element `bold` or element `italic`. The asterisk indicates that the content can occur zero or more times. Lines 8 and 9 specify that `bold` and `italic` elements only have `PCDATA` for their content specification—they cannot contain child elements. Despite the fact that elements with `PCDATA` content specification cannot contain child elements, they are still considered to have mixed content. The comma (,), plus sign (+) and question mark (?) occurrence indicators cannot be used with mixed-content elements that contain only `PCDATA`.

Figure B.6 shows the results of changing the first pipe character in line 7 of Fig. B.5 to a comma and the result of removing the asterisk. Both of these are illegal DTD syntax.

**Common Programming Error B.3**

When declaring mixed content, not listing `PCDATA` as the first item is an error.

An element declared as type `ANY` can contain any content, including `PCDATA`, elements or a combination of elements and `PCDATA`. Elements with `ANY` content can also be empty elements.

**Software Engineering Observation B.3**

Elements with `ANY` content are commonly used in the early stages of DTD development. Document authors typically replace `ANY` content with more specific content as the DTD evolves.

## B.5 Attribute Declarations

In this section, we discuss attribute declarations. An attribute declaration specifies an attribute list for an element by using the `ATTLIST` attribute list declaration. An element can have any number of attributes. For example,

```
<!ELEMENT x EMPTY>
<!ATTLIST x y CDATA #REQUIRED>
```

```xml
<?xml version = "1.0" standalone = "yes"?>
<!-- Fig. B.6 : invalid-mixed.xml -->
<!-- Mixed content type elements -->
<!DOCTYPE format [
  <!ELEMENT format ( #PCDATA | bold, italic )>
  <!ELEMENT bold ( #PCDATA )>
  <!ELEMENT italic ( #PCDATA )>
]>

<format>
  Book catalog entry:
  <bold>XML</bold>
  <italic>XML How to Program</italic>
  This book carefully explains XML-based systems development.
</format>
```

**Fig. B.6** Changing a pipe character to a comma in a DTD (part 1 of 2).
Appendix B  Document Type Definition (DTD)  1637

declares **EMPTY** element `x`. The attribute declaration specifies that `y` is an attribute of `x`. Keyword **CDATA** indicates that `y` can contain any character text except for the `<, >, &,' and " characters. Note that the **CDATA** keyword in an attribute declaration has a different meaning than the **CDATA** section in an XML document we introduced in Appendix A. Recall that in a **CDATA** section all characters are legal except the `]>` end tag. **Keyword #REQUIRED** specifies that the attribute must be provided for element `x`. We will say more about other keywords momentarily.

Figure B.7 demonstrates how to specify attribute declarations for an element. Line 9 declares attribute `id` for element `message`. Attribute `id` contains required **CDATA**. Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character). We discuss normalization in detail in Section B.8. Line 13 assigns attribute `id` the value "6343070".

DTDs allow document authors to specify an attribute’s default value using **attribute defaults**, which we briefly touched upon in the previous section. Keywords **#IMPLIED**, **#REQUIRED** and **#FIXED** are attribute defaults. Keyword **#IMPLIED** specifies that if the attribute does not appear in the element, then the application using the XML document can use whatever value (if any) it chooses.

Keyword **#REQUIRED** indicates that the attribute must appear in the element. The XML document is not valid if the attribute is missing. For example, the markup

```xml
<message>XML and DTDs</message>
```

---

**Fig. B.6**  Changing a pipe character to a comma in a DTD (part 2 of 2).

```bash
C:>java -jar Validator.jar invalid-mixed.xml
fatal error: Mixed content model for "format" must end with ")*", not ",".
```

**Fig. B.7**  Declaring attributes (part 1 of 2).

```xml
<?xml version = "1.0"?>
<!-- Fig. B.7: welcome2.xml -->
<!-- Declaring attributes -->
<!DOCTYPE myMessage [ 
<!ELEMENT myMessage ( message )>
<!ELEMENT message ( #PCDATA )>
<!ATTLIST message id CDATA #REQUIRED>
]>

<myMessage>
  <message id = "6343070">
    Welcome to XML!
  </message>
</myMessage>
```
when checked against the DTD attribute list declaration

```
<!ATTLIST message number CDATA #REQUIRED>
```
does not conform to it because attribute `number` is missing from element `message`.

An attribute declaration with default value `#FIXED` specifies that the attribute value is constant and cannot be different in the XML document. For example,

```
<!ATTLIST address zip #FIXED "02115">
```

indicates that the value "02115" is the only value attribute `zip` can have. The XML document is not valid if attribute `zip` contains a value different from "02115". If element `address` does not contain attribute `zip`, the default value "02115" is passed to the application that is using the XML document’s data.

### B.6 Attribute Types

Attribute types are classified as either string (`CDATA`), tokenized or enumerated. String attribute types do not impose any constraints on attribute values, other than disallowing the `<` and `&` characters. Entity references (e.g., `&lt;`, `&amp;`, etc.) must be used for these characters. Tokenized attribute types impose constraints on attribute values, such as which characters are permitted in an attribute name. We discuss tokenized attribute types in the next section. Enumerated attribute types are the most restrictive of the three types. They can take only one of the values listed in the attribute declaration. We discuss enumerated attribute types in Section B.6.2.

#### B.6.1 Tokenized Attribute Type (`ID`, `IDREF`, `ENTITY`, `NMTOKEN`)

Tokenized attribute types allow a DTD author to restrict the values used for attributes. For example, an author may want to have a unique ID for each element or allow an attribute to have only one or two different values. Four different tokenized attribute types exist: `ID`, `IDREF`, `ENTITY` and `NMTOKEN`.

Tokenized attribute type `ID` uniquely identifies an element. Attributes with type `IDREF` point to elements with an `ID` attribute. A validating parser verifies that every `ID` attribute type referenced by `IDREF` is in the XML document.

Figure B.8 lists an XML document that uses `ID` and `IDREF` attribute types. Element `bookstore` consists of element `shipping` and element `book`. Each `shipping` element describes who shipped the book and how long it will take for the book to arrive.

Line 9 declares attribute `shipID` as an `ID` type attribute (i.e., each `shipping` element has a unique identifier). Lines 27–37 declare `book` elements with attribute `shippedBy` (line 11) of type `IDREF`. Attribute `shippedBy` points to one of the `shipping` elements by matching its `shipID` attribute.
Common Programming Error B.4

Using the same value for multiple ID attributes is a logic error: The document validated against the DTD is not valid.

The DTD contains an entity declaration for each of the entities isbnXML, isbnJava and isbnCPP. The parser replaces the entity references with their values. These entities are called general entities.

Figure B.9 is a variation of Fig. B.8 that assigns shippedBy (line 32) the value "bug". No shipID attribute has a value "bug", which results in a invalid XML document.

```xml
<?xml version = "1.0"?>
<!DOCTYPE bookstore [
<!ELEMENT bookstore ( shipping+, book+ )>
<!ELEMENT shipping ( duration )>
<!ATTLIST shipping shipID ID #REQUIRED>
<!ELEMENT book ( #PCDATA )>
<!ATTLIST book shippedBy IDREF #IMPLIED>
<!ELEMENT duration ( #PCDATA )>
<!ENTITY isbnXML "0-13-028417-3">
<!ENTITY isbnJava "0-13-034151-7">
<!ENTITY isbnCPP "0-13-0895717-3">
]>    
<bookstore>
  <shipping shipID = "bug2bug">
    <duration>2 to 4 days</duration>
  </shipping>
  <shipping shipID = "Deitel">
    <duration>1 day</duration>
  </shipping>
  <book shippedBy = "Deitel" isbn = "&isbnJava;">
    Java How to Program 4th edition.
  </book>
  <book shippedBy = "Deitel" isbn = "&isbnXML;">
    XML How to Program.
  </book>
  <book shippedBy = "bug2bug" isbn = "&isbnCPP;">
    C++ How to Program 3rd edition.
  </book>
</bookstore>
```

C:\>java -jar Validator.jar IDExample.xml
Document is valid.

Fig. B.8 XML document with ID and IDREF attribute types (part 1 of 2).
Common Programming Error B.5
Not beginning a type attribute ID's value with a letter, an underscore (_) or a colon (:) is an error.

Common Programming Error B.6
Providing more than one ID attribute type for an element is an error.

Common Programming Error B.7
Declaring attributes of type ID as #FIXED is an error.

Related to entities are entity attributes, which indicate that an attribute has an entity for its value. Entity attributes are specified by using tokenized attribute type ENTITY. The primary constraint placed on ENTITY attribute types is that they must refer to external unparsed entities. An external unparsed entity is defined in the external subset of a DTD and consists of character data that will not be parsed by the XML parser.

Figure B.10 lists an XML document that demonstrates the use of entities and entity attribute types.
Line 7 declares a *notation* named `html` that refers to a *SYSTEM* identifier named "*iexplorer". Notations provide information that an application using the XML document can use to handle unparsed entities. For example, the application using this document may choose to open Internet Explorer and load the document `tour.html` (line 8).

Line 8 declares an entity named `city` that refers to an external document (`tour.html`). *Keyword NDATA* indicates that the content of this external entity is not XML. The name of the notation (e.g., `html`) that handles this unparsed entity is placed to the right of `NDATA`.

Line 11 declares attribute `tour` for element `company`. Attribute `tour` specifies a required *ENTITY* attribute type. Line 16 assigns entity `city` to attribute `tour`. If we replaced line 16 with
the document fails to conform to the DTD because entity **country** does not exist. Figure B.11 shows the error message generated when the above replacement is made.

```xml
<!DOCTYPE database [ 
<!NOTATION xhtml SYSTEM "iexplorer"> 
<!ENTITY city SYSTEM "tour.html" NDATA xhtml> 
<!ELEMENT database ( company+ )> 
<!ELEMENT company ( name )> 
<!ATTLIST company tour ENTITY #REQUIRED> 
<!ELEMENT name ( #PCDATA )> ]>
<database>
  <company tour = "country">
    <name>Deitel & Associates, Inc.</name>
  </company>
</database>
```

```
C:\>java -jar Validator.jar entityexample.xml
Document is valid.
```

**Fig. B.10**  XML document that contains an **ENTITY** attribute type.

```xml
<!DOCTYPE database [ 
<!NOTATION xhtml SYSTEM "iexplorer"> 
<!ENTITY city SYSTEM "tour.html" NDATA xhtml> 
<!ELEMENT database ( company+ )> 
<!ELEMENT company ( name )> 
<!ATTLIST company tour ENTITY #REQUIRED> 
<!ELEMENT name ( #PCDATA )> ]>
<database>
  <company tour = "city">
    <name>Deitel & Associates, Inc.</name>
  </company>
</database>
```

```
C:\>java -jar Validator.jar invalid-entityExample.xml
```

**Fig. B.11**  Error generated when a DTD contains a reference to an undefined entity (part 1 of 2).
Appendix B  Document Type Definition (DTD)  1643

Common Programming Error B.8
Not assigning an unparsed external entity to an attribute with attribute type ENTITY results in an invalid XML document.

Attribute type ENTITIES may also be used in a DTD to indicate that an attribute has multiple entities for its value. Each entity is separated by a space. For example

```xml
<!ATTLIST directory file ENTITIES #REQUIRED>
```
specifies that attribute file is required to contain multiple entities. An example of markup that conforms to this might look like

```xml
<directory file = "animations graph1 graph2">
```
where animations, graph1 and graph2 are entities declared in a DTD.
A more restrictive attribute type is NMTOKEN (name token), whose value consists of letters, digits, periods, underscores, hyphens and colon characters. For example, consider the declaration

```xml
<!ATTLIST sportsClub phone NMTOKEN #REQUIRED>
```
which indicates sportsClub contains a required NMTOKEN phone attribute. An example of markup that conforms to this is

```xml
<sportsClub phone = "555-111-2222">
```
An example that does not conform to this is

```xml
<sportsClub phone = "555 555 4902">
```
because spaces are not allowed in an NMTOKEN attribute.
Similarly, when an NMTOKENS attribute type is declared, the attribute may contain multiple string tokens separated by spaces.

B.6.2 Enumerated Attribute Types
Enumerated attribute types declare a list of possible values an attribute can have. The attribute must be assigned a value from this list to conform to the DTD. Enumerated type values are separated by pipe characters ( | ). For example, the declaration

```xml
<!ATTLIST person gender ( M | F ) "F">
```
contains an enumerated attribute type declaration that allows attribute gender to have either the value M or the value F. A default value of "F" is specified to the right of the element attribute type. Alternatively, a declaration such as

```xml
<!ATTLIST person gender ( M | F ) #IMPLIED>  
```
does not provide a default value for gender. This type of declaration might be used to validate a marked-up mailing list that contains first names, last names, addresses, etc. The application that uses such a mailing list may want to precede each name by either Mr., Ms. or Mrs. However, some first names are gender neutral (e.g., Chris, Sam, etc.), and the application may not know the person’s gender. In this case, the application has the flexibility to process the name in a gender-neutral way.

NOTATION is also an enumerated attribute type. For example, the declaration

```xml
<!ATTLIST book reference NOTATION ( JAVA | C ) "C">
```

indicates that reference must be assigned either JAVA or C. If a value is not assigned, C is specified as the default. The notation for C might be declared as

```xml
<!NOTATION C SYSTEM
 "http://www.deitel.com/books/2000/chtp3/chtp3_toc.htm">
```

### B.7 Conditional Sections

DTDs provide the ability to include or exclude declarations using conditional sections. Keyword INCLUDE specifies that declarations are included, while keyword IGNORE specifies that declarations are excluded. For example, the conditional section

```xml
<!INCLUDE[
  <!ELEMENT name ( #PCDATA )>
]>]
```

directs the parser to include the declaration of element name.

Similarly, the conditional section

```xml
<!IGNORE[
  <!ELEMENT message ( #PCDATA )>
]>]
```

directs the parser to exclude the declaration of element message. Conditional sections are often used with entities, as demonstrated in Fig. B.12.

```xml
<!ENTITY % reject "IGNORE">
<!ENTITY % accept "INCLUDE">

<![ %accept; [
  <!ELEMENT message ( approved, signature )>
]>]

<![ %reject; [
  <!ELEMENT message ( approved, reason, signature )>
]>]

<!ELEMENT approved EMPTY>
<!ATTLIST approved flag ( true | false ) "false">
```

---

**Fig. B.12** Conditional sections in a DTD (part 1 of 2).
Lines 4–5 declare entities reject and accept, with the values IGNORE and INCLUDE, respectively. Because each of these entities is preceded by a percent (%) character, they can be used only inside the DTD in which they are declared. These types of entities—called parameter entities—allow document authors to create entities specific to a DTD—not an XML document. Recall that the DTD is the combination of the internal subset and external subset. Parameter entities may be placed only in the external subset.

Lines 7–13 use the entities accept and reject, which represent the strings INCLUDE and IGNORE, respectively. Notice that the parameter entity references are preceded by %, whereas normal entity references are preceded by &. Line 7 represents the beginning tag of an IGNORE section (the value of the accept entity is IGNORE), while line 11 represents the start tag of an INCLUDE section. By changing the values of the entities, we can easily choose which message element declaration to allow.

Figure B.13 shows the XML document that conforms to the DTD in Fig. B.12.

Software Engineering Observation B.4

Parameter entities allow document authors to use entity names in DTDs without conflicting with entities names used in an XML document.

B.8 Whitespace Characters

In Appendix A, we briefly discussed whitespace characters. In this section, we discuss how whitespace characters relate to DTDs. Depending on the application, insignificant whitespace characters may be collapsed into a single whitespace character or even removed entirely. This process is called normalization. Whitespace is either preserved or normalized, depending on the context in which it is used.
Figure B.14 contains a DTD and markup that conforms to the DTD. Line 28 assigns a value containing multiple whitespace characters to attribute `cdata`. Attribute `cdata` (declared in line 11) is required and must contain `CDATA`. As mentioned earlier, `CDATA` can contain almost any text, including whitespace. As the output illustrates, spaces in `CDATA` are preserved and passed on to the application that is using the XML document.

Line 30 assigns a value to attribute `id` that contains leading whitespace. Attribute `id` is declared on line 14 with tokenized attribute type `ID`. Because this is not `CDATA`, it is normalized and the leading whitespace characters are removed. Similarly, lines 32 and 34 assign values that contain leading whitespace to attributes `nmtoken` and `enumeration`—which are declared in the DTD as an `NMTOKEN` and an enumeration, respectively. Both these attributes are normalized by the parser.
This is some additional text.
</hasMixed>

This is text.
  <hasCDATA cdata=" simple cdata "/>
  <hasID id="i20"/>
  <hasNMTOKEN nmtoken="hello"/>
  <hasEnumeration enumeration="true"/>
</hasMixed>

This is some additional text.
</whitespace>

Fig. B.14  Processing whitespace in an XML document (part 2 of 2).

B.9 Internet and World Wide Web Resources

www.wdvl.com/Authoring/HTML/Validation/DTD.html
Contains a description of the historical uses of DTDs, including a description of SGML and the HTML DTD.

www.xml101.com/dtd
Contains tutorials and explanations on creating DTDs.

www.w3schools.com/dtd
Contains DTD tutorials and examples.

A DTD tutorial for Web developers.

www.schema.net
A DTD repository with XML links and resources.

www.networking.ibm.com/xml/XmlValidatorForm.html
IBM’s DOMit XML Validator.
**SUMMARY**

- Document Type Definitions (DTDs) define an XML document’s structure (e.g., what elements, attributes, etc. are permitted in the XML document). An XML document is not required to have a corresponding DTD. DTDs use EBNF (Extended Backus-Naur Form) grammar.

- Parsers are generally classified as validating or nonvalidating. A validating parser can read the DTD and determine whether or not the XML document conforms to it. If the document conforms to the DTD, it is referred to as valid. If the document fails to conform to the DTD but is syntactically correct, it is well formed but not valid. By definition, a valid document is well formed.

- A nonvalidating parser is able to read a DTD, but cannot check the document against the DTD for conformity. If the document is syntactically correct, it is well formed.

- DTDs are introduced into XML documents by using the document type declaration (i.e., `DOCTYPE`). The document type declaration can point to declarations that are outside the XML document (called the external subset) or can contain the declaration inside the document (called the internal subset).

- External subsets physically exist in a different file that typically ends with the `.dtd` extension, although this file extension is not required. External subsets are specified using keyword `SYSTEM` or `PUBLIC` Both the internal and external subset may be specified at the same time.

- Elements are the primary building block used in XML documents and are declared in a DTD with element type declarations (`ELEMENTs`).

- The element name that follows `ELEMENT` is often called a generic identifier. The set of parentheses that follow the element name specify the element’s allowed content and is called the content specification.

- Keyword `PCDATA` specifies that the element must contain parsable character data—that is, any text except the characters less than (`<`) and ampersand (`&`).

- An XML document is a standalone XML document if it does not reference an external DTD.

- An XML element that can have only another element for content is said to have element content.

- DTDs allow document authors to define the order and frequency of child elements. The comma (`,`)—called a sequence—specifies the order in which the elements must occur. Choices are specified using the pipe character (`|`). The content specification may contain any number of pipe-character-separated choices.

- An element’s frequency (i.e., number of occurrences) is specified by using either the plus sign (`+`), asterisk (`*`) or question mark (`?`) occurrence indicator.

- The frequency of an element group (i.e., two or more elements that occur in some combination) is specified by enclosing the element names inside the content specification, followed by an occurrence indicator.

- Elements can be further refined by describing the content types they may contain. Content specification types (e.g., `EMPTY`, mixed content, `ANY`, etc.) describe nonelement content.

- An element can be declared as having mixed content (i.e., a combination of elements and `PCDATA`). The comma (`,`), plus sign (`+`) and question mark (`?`) occurrence indicators cannot be used with mixed content elements.

- An element declared as type `ANY` can contain any content including `PCDATA`, elements, or a combination of elements and `PCDATA`. Elements with `ANY` content can also be empty elements.

- An attribute list for an element is declared using the `ATTLIST` element type declaration.

- Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character).
• DTDs allow document authors to specify an attribute’s default value using attribute defaults. Keywords #IMPLIED, #REQUIRED and #FIXED are attribute defaults.

• Keyword #IMPLIED specifies that if the attribute does not appear in the element, then the application using the XML document can apply whatever value (if any) it chooses.

• Keyword #REQUIRED indicates that the attribute must appear in the element. The XML document is not valid if the attribute is missing.

• An attribute declaration with default value #FIXED specifies that the attribute value is constant and cannot be different in the XML document.

• Attribute types are classified as either string (CDATA), tokenized or enumerated. String attribute types do not impose any constraints on attribute values, other than disallowing the < and & characters. Entity references (e.g., &lt;, &amp;, etc.) must be used for these characters. Tokenized attributes impose constraints on attribute values, such as which characters are permitted in an attribute name. Enumerated attributes are the most restrictive of the three types. They can take only one of the values listed in the attribute declaration.

• Four different tokenized attribute types exist: ID, IDREF, ENTITY and NMTOKEN. Tokenized attribute type ID uniquely identifies an element. Attributes with type IDREF point to elements with an ID attribute. A validating parser verifies that every ID attribute type referenced by IDREF is in the XML document.

• Entity attributes indicate that an attribute has an entity for its value and are specified using tokenized attribute type ENTITY. The primary constraint placed on ENTITY attribute types is that they must refer to external unparsed entities.

• Attribute type ENTITIES may also be used in a DTD to indicate that an attribute has multiple entities for its value. Each entity is separated by a space.

• A more restrictive attribute type is attribute type NMTOKEN (name token), whose value consists of letters, digits, periods, underscores, hyphens and colon characters.

• Attribute type NMTOKENS may contain multiple string tokens separated by spaces.

• Enumerated attribute types declare a list of possible values an attribute can have. The attribute must be assigned a value from this list to conform to the DTD. Enumerated type values are separated by pipe characters (|).

• NOTATION is also an enumerated attribute type. Notations provide information that an application using the XML document can use to handle unparsed entities.

• Keyword NDATA indicates that the content of an external entity is not XML. The name of the notation that handles this unparsed entity is placed to the right of NDATA.

• DTDs provide the ability to include or exclude declarations using conditional sections. Keyword INCLUDE specifies that declarations are included, while keyword IGNORE specifies that declarations are excluded. Conditional sections are often used with entities.

• Parameter entities are preceded by percent (%) characters and can be used only inside the DTD in which they are declared. Parameter entities allow document authors to create entities specific to a DTD—not an XML document.

• Whitespace is either preserved or normalized, depending on the context in which it is used. Spaces in CDATA are preserved. Attribute values with tokenized attribute types ID, NMTOKEN and enumeration are normalized.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>#FIXED</th>
<th>#PCDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>#IMPLIED</td>
<td>#REQUIRED</td>
</tr>
</tbody>
</table>
SELF-REVIEW EXERCISES

B.1 State whether the following are true or false. If the answer is false, explain why.
a) The document type declaration, **DOCTYPE**, introduces DTDs in XML documents.
b) External DTDs are specified by using the keyword **EXTERNAL**.
c) A DTD can contain either internal or external subsets of declarations, but not both.
d) Child elements are declared in parentheses, inside an element type declaration.
e) An element that appears any number of times is followed by an exclamation point (!).
f) A mixed-content element can contain text as well as other declared elements.
g) An attribute declared as type CDATA can contain all characters except for the asterisk (*) and pound sign (#).
h) Each element attribute of type ID must have a unique value.
i) Enumerated attribute types are the most restrictive category of attribute types.
j) An enumerated attribute type requires a default value.

B.2 Fill in the blanks in each of the following statements:
a) The set of document type declarations inside an XML document is called the __________.
b) Elements are declared with the __________ type declaration.
c) Keyword __________ indicates that an element contains parsable character data.
d) In an element type declaration, the pipe character (|) indicates that the element can contain __________ of the elements indicated.
e) Attributes are declared by using the __________ type.
f) Keyword __________ specifies that the attribute can take only a specific value that has been defined in the DTD.
g) ID, IDREF, __________ and NM_TOKEN are all types of tokenized attributes.
h) The % character is used to declare a/an __________.
i) DTD is an acronym for __________.
j) Conditional sections of DTDs are often used with __________.

ANSWERS TO SELF-REVIEW EXERCISES

B.1 a) True. b) False. External DTDs are specified using keyword SYSTEM. c) False. A DTD contains both the internal and external subsets. d) True. e) False. An element that appears one or zero times is specified by a question mark (?). f) True. g) False. An attribute declared as type CDATA can contain all characters except for ampersand (&), less than (<), greater than (>), quote (') and double quotes ("). h) True. i) True. j) False. A default value is not required.

B.2 a) internal subset. b) ELEMENT. c) PCDATA. d) one. e) ATTLIST. f) #FIXED. g) ENTITY. h) parameter entity. i) Document Type Definition. j) entities.
Objectives

• To understand what the Document Object Model is.
• To understand and be able to use the major DOM features.
• To use Java to manipulate an XML document.
• To become familiar with DOM-based parsers.

Knowing trees, I understand the meaning of patience.
Knowing grass, I can appreciate persistence.
Hal Borland

There was a child went forth every day,
And the first object he look’d upon, that object he became.
Walt Whitman

I think that I shall never see
A poem lovely as a tree.
Joyce Kilmer
Appendix C

C.1 Introduction

In previous appendices, we concentrated on basic XML markup and DTDs for validating XML documents. In this appendix, we focus on manipulating the contents of an XML document.

XML documents, when parsed, are represented as a hierarchical tree structure in memory. This tree structure contains the document’s elements, attributes, content, etc. XML was designed to be a live, dynamic technology—a programmer can modify the contents of the tree structure, which essentially allows the programmer to add data, remove data, query for data, etc. in a manner similar to manipulating a database.

The W3C provides a recommendation for building a tree structure in memory for XML documents. This structure is called the XML Document Object Model (DOM). Any parser that adheres to this recommendation is called a DOM-based parser. Each element, attribute, CDATA section, etc., in an XML document is represented by a node in the DOM tree. For example, the simple XML document

```
<?xml version = "1.0"?>
<message from = "Paul" to = "Tem">
  <body>Hi, Tem!</body>
</message>
```

results in a DOM tree with several nodes. One node is created for the message element. This node has a child node that corresponds to the body element. The body element has a child node that corresponds to the text Hi, Tem!. The from and to attributes of the message element also have corresponding nodes in the DOM tree as well. An XML declaration is not placed in the tree.

A DOM-based parser exposes (i.e., makes available) a programmatic library—called the DOM Application Programming Interface (API)—that allows data in an XML document to be accessed and modified by manipulating the nodes in a DOM tree. In this appendix, we use Sun Microsystem’s JAXP parsers.

Portability Tip C.1

The DOM interfaces for creating and manipulating XML documents are platform and language independent. DOM parsers exist for many different languages, including Java, C, C++, Python and Perl.
C.2 DOM with Java

To introduce document manipulation with the XML Document Object Model, we begin with a simple example that uses Java. This example takes an XML document (Fig. C.1) that marks up an article and uses the JAXP API to display the document’s element names and values. Figure C.2 lists the Java code that manipulates this XML document and displays its content.

```xml
<?xml version = "1.0"?>
<!-- Fig. C.1: article.xml -->
<!-- Article formatted with XML -->
<article>
  <title>Simple XML</title>
  <date>July 31, 2001</date>
  <author>
    <fname>Tem</fname>
    <lname>Nieto</lname>
  </author>
  <summary>XML is easy.</summary>
  <content>Once you have mastered XHTML, you can easily learn XML. You must remember that XML is not for displaying information but for managing information.</content>
</article>
```

Fig. C.1 Article marked up with XML tags.

```java
// Fig C.2 : XMLInfo.java
// Outputs node information
// Java core libraries
import java.io.*;
// Java standard extensions
import javax.xml.parsers.*;
// third-party libraries
import org.w3c.dom.*;
import org.xml.sax.*;

public class XMLInfo {

Fig. C.2 XMLInfo displays information about XML input (part 1 of 3).
```
```java
public static void main( String args[] )
{
    if ( args.length != 1 ) {
        System.err.println( "Usage: java XMLInfo input.xml" ) ;
        System.exit( 1 );
    }
	ry {

// create DocumentBuilderFactory
    DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();

// create DocumentBuilder
    DocumentBuilder builder = factory.newDocumentBuilder();

// obtain document object from XML document
    Document document = builder.parse( new File( args[ 0 ] ) );

// get root node
    Node root = document.getDocumentElement();

System.out.print( "Here is the document's root node:" );
    System.out.println( " " + root.getNodeName() );

System.out.println( "Here are its child elements: " );
    NodeList childNodes = root.getChildNodes();
    Node currentNode;

    for ( int i = 0; i < childNodes.getLength(); i++ ) {
        currentNode = childNodes.item( i );

        // print node name of each child element
        System.out.println( currentNode.getNodeName() );
    }

// get first child of root element
    currentNode = root.getFirstChild();

System.out.print( "The first child of root node is: " );
    System.out.println( currentNode.getNodeName() );

// get next sibling of first child
    System.out.print( "whose next sibling is: " );
    currentNode = currentNode.getNextSibling();
    System.out.println( currentNode.getNodeName() );

// print value of first child’s next sibling
    System.out.println( "value of " + currentNode.getNodeName() + " element is: " +
```
import several packages related to XML. Package `javax.xml.parsers` provides classes related to parsing an XML document. Package `org.w3c.dom` provides the DOM-API programmatic interface (i.e., classes, methods, etc.).

Lines 27–28 create a new `DocumentBuilderFactory`. The `DocumentBuilderFactory` is required to produce an appropriate `DocumentBuilder` object for the currently configured XML parser. JAXP can be configured to use many different XML parsers, such as the Apache Group’s Xerces and IBM’s XML4J. JAXP also has its own parser built in, which is used by default.

Line 38 retrieves the root node of the XML document via `Document` method `getElementsByTagName`. Line 41 retrieves and displays the name of the `Node` via method `getNodeName`. Line 44 calls `Node` method `getElementsByTagName` to obtain a `NodeList` object, which is list of nodes. The first item added is stored at index 0, the next at index 1, and so forth. This index is used to access an individual item in the `NodeList`.

Line 47 uses `NodeList` method `getLength` to obtain the number of nodes in the list. Lines 47–53 display the name of each `Node` in the `NodeList`, by calling `NodeList` method `item`. This method is passed the index of the desired `Node` in the `NodeList`.

Line 56 calls method `getFirstChild` to obtain a `Node` reference to the first child of `Node root`. Line 59 displays the name of the `Node` stored in `currentNode`. Line 63 calls `Node` method `getNextSibling` to obtain a reference to the next sibling of the node. Line 64 displays the sibling `Node`’s name. Lines 67–69 print the value of the first child of `currentNode`. The `Node` method `getNodeValue` returns different objects for different types of nodes. In our XML document (Fig. C.1), the child node happens to be of type `text`, so `getNodeValue` returns the `String` contents of the `text` node. We will explain `Node` types in greater detail later in this appendix. Lines 78–92 catch exceptions that various methods we have used throw.

### C.3 Setup Instructions

To manipulate XML within Java, you must properly install the Java API for XML Processing (JAXP). To do so, complete the following steps:

1. Download and run the JAXP 1.1 installer from [www.java.sun.com/xml](http://www.java.sun.com/xml)

2. Copy `jaxp.jar, crimson.jar, xalan.jar` from the `JAXP1.1` directory to the `C:\jdk1.3.1\jre\lib\ext\` directory.

3. Add `C:\jdk1.3.1\jre\bin` to the `PATH` environment variable before `C:\jdk1.3.1\bin`.

4. Run each example as you normally would.

As we present an example, we will discuss any special steps necessary to execute it. However the steps outlined in this section must be followed before attempting to execute any example.

### C.4 DOM Components

In this section, we will use Java, JAXP and interfaces described in Fig. C.3 to manipulate an XML document. Due to the number of DOM interfaces and methods available, we provide only a partial list.

For a complete list of DOM classes and interfaces, browse the HTML documentation (`index.html` in the `api` folder) included with JAXP.
The **Document** interface represents the top-level node of an XML document in memory and provides a means of creating nodes and retrieving nodes. Figure C.4 lists some **Document** methods.

Interface **Node** represents an XML document node. Figure C.5 lists the methods of interface **Node**.

Figure C.6 lists some node types that may be returned by method **getNodeType**. Each type in Fig. C.6 is a **static final** (i.e., constant) member of class **Node**.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Document</strong> interface</td>
<td>Represents the XML document’s top-level node, which provides access to all the document’s nodes, including the root element.</td>
</tr>
<tr>
<td><strong>Node</strong> interface</td>
<td>Represents an XML document node.</td>
</tr>
<tr>
<td><strong>NodeList</strong> interface</td>
<td>Represents a read-only list of <strong>Node</strong> objects.</td>
</tr>
<tr>
<td><strong>Element</strong> interface</td>
<td>Represents an element node. Derives from <strong>Node</strong>.</td>
</tr>
<tr>
<td><strong>Attr</strong> interface</td>
<td>Represents an attribute node. Derives from <strong>Node</strong>.</td>
</tr>
<tr>
<td><strong>CharacterData</strong> interface</td>
<td>Represents character data. Derives from <strong>Node</strong>.</td>
</tr>
<tr>
<td><strong>Text</strong> interface</td>
<td>Represents a text node. Derives from <strong>CharacterData</strong>.</td>
</tr>
<tr>
<td><strong>Comment</strong> interface</td>
<td>Represents a comment node. Derives from <strong>CharacterData</strong>.</td>
</tr>
<tr>
<td><strong>ProcessingInstruction</strong></td>
<td>Represents a processing instruction node. Derives from <strong>Node</strong>.</td>
</tr>
<tr>
<td><strong>CDATASection</strong> interface</td>
<td>Represents a <strong>CDATA</strong> section. Derives from <strong>Text</strong>.</td>
</tr>
</tbody>
</table>

**Fig. C.3**   DOM classes and interfaces.

**Fig. C.4**   Some **Document** methods.
### Method Name | Description
---|---
appendChild | Appends a child node.
cloneNode | Duplicates the node.
getAttributes | Returns the node’s attributes.
getChildNodes | Returns the node’s child nodes.
getNextSibling | Returns the node’s next sibling.
getName | Returns the node’s name.
getType | Returns the node’s type (e.g., element, attribute, text, etc.). Node types are described in greater detail in Fig. C.6.
getNodeValue | Returns the node’s value.
getParentNode | Returns the node’s parent.
hasChildNodes | Returns true if the node has child nodes.
removeChild | Removes a child node from the node.
replaceChild | Replaces a child node with another node.
setNodeValue | Sets the node’s value.
insertBefore | Appends a child node in front of a child node.

**Fig. C.5** Node methods.

### Node Type | Description
---|---
Node.ELEMENT_NODE | Represents an element node.
Node.ATTRIBUTE_NODE | Represents an attribute node.
Node.TEXT_NODE | Represents a text node.
Node.COMMENT_NODE | Represents a comment node.
Node.PROCESSING_INSTRUCTION_NODE | Represents a processing instruction node.
Node.CDATA_SECTION_NODE | Represents a CDATA section node.

**Fig. C.6** Some node types.

*Element* represents an element node. Figure C.7 lists some *Element* methods.

### Method Name | Description
---|---
getAttribute | Returns the value of the attribute with the given name.
getTagName | Returns an element’s name.

**Fig. C.7** Element methods (part 1 of 2).
removeAttribute
Sets an attribute’s value.

**Fig. C.7**  **Element** methods (part 2 of 2).

Figure C.8 lists a Java application that validates *intro.xml* (Fig. C.10) and replaces the text in its *message* element with *New Changed Message!!*.

```java
// Fig C.8 : ReplaceText.java
// Reads intro.xml and replaces a text node.
// Java core packages
import java.io.*;

// Java extension packages
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;

// third-party libraries
import org.xml.sax.*;
import org.w3c.dom.*;

public class ReplaceText {
    private Document document;

    public ReplaceText() {
        // parse document, find/replace element, output result
        try {

            // obtain default parser
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();

            // set parser as validating
            factory.setValidating( true );

            // obtain object that builds Documents
            DocumentBuilder builder = factory.newDocumentBuilder();

            // set error handler for validation errors
            builder.setErrorHandler( new MyErrorHandler() );

            // obtain document object from XML document
            document = builder.parse( new File( "c:/intro.xml" ) );
        }
    }
}
```

**Fig. C.8**  Simple example that replaces an existing text node (part 1 of 3).
// retrieve the root node
Node root = document.getDocumentElement();

if ( root.getNodeType() == Node.ELEMENT_NODE ) {
    Element myMessageNode = ( Element ) root;
    NodeList messageNodes =
        myMessageNode.getElementsByTagName( "message" );

    if ( messageNodes.getLength() != 0 ) {
        Node message = messageNodes.item( 0 );

        // create text node
        Text newText = document.createTextNode( 
            "New Changed Message!!" );

        // get old text node
        Text oldText =
            ( Text ) message.getChildNodes().item( 0 );

        // replace text
        message.replaceChild( newText, oldText );
    }
}

// output Document object

// create DOMSource for source XML document
Source xmlSource = new DOMSource( document );

// create StreamResult for transformation result
Result result = new StreamResult( System.out );

// create TransformerFactory
TransformerFactory transformerFactory =
    TransformerFactory.newInstance();

// create Transformer for transformation
Transformer transformer =
    transformerFactory.newTransformer();

transformer.setOutputProperty( "indent", "yes" );

// transform and deliver content to client
transformer.transform( xmlSource, result );

// handle exception creating DocumentBuilder
catch ( ParserConfigurationException parserException ) {
    parserException.printStackTrace();
}

// handle exception parsing Document
Lines 27–28 create a new `DocumentBuilderFactory`. Line 31 indicates that a validating parser should be used by passing the value `true` as an argument to method `setValidating`. Line 34 creates a new `DocumentBuilder`. Line 37 specifies that a `MyErrorHandler` (Fig. C.9) object provides methods for handling exceptions related to parsing. Line 40 calls method `parse` to load and parse the XML document stored in the file `intro.xml`. If parsing is successful, a `Document` object is returned that contains nodes representing each part of the `intro.xml` document. If parsing fails, a `SAXException` is thrown.

Line 43 gets the `Document`’s root node. Line 45 calls method `getNodeType` to obtain the root node’s type and tests whether the root node is of type element.

Line 46 downcasts `root` from a superclass `Node` type to an `Element` derived type. As mentioned earlier, class `Element` inherits from class `Node`. Lines 47–48 get a `NodeList` containing all the `message` elements in the XML document using method `getElementsByTagName`. Line 50 determines whether the `NodeList` contains at least one item. Line 51 retrieves the first `Node` in the `NodeList`.

Lines 54–55 call `createTextNode` to create a text node that contains the text `New Changed Message!!`. This node exists in memory independent of the XML docu-
ment—i.e., it has not been inserted into the document yet. Interface `Text` represents an element or attribute’s character data.

Lines 58–59 get the first child node of the `message` element (referred by `Node message` in line 51), which is a text node containing the text `Welcome to XML!`. Method `item` returns an object of superclass type `Object` which we downcast to `Text`. Line 62 calls method `replaceChild` to replace the `Node` referenced by the second argument with the `Node` referenced by the first argument. The XML document has now been modified—element `message` now contains the text `New Changed Message!!`.

Lines 66–86 output the modified `Document`. Line 69 creates a new `Source` object that wraps the modified `Document` object. Line 72 creates a new `Result` object passing an `OutputStream` as an argument, which in this case is `System.out`. Lines 75–76 create a new `TransformerFactory` by calling static method `newInstance`. Lines 79–80 create a new `Transformer` by calling method `newTransformer`.

Line 82 sets the "indent" output property to "yes," which causes the `Transformer` to add appropriate indentation when it produces the result document. Lines 89-90 handle a `ParserConfigurationException`, which can be thrown by `DocumentBuilderFactory` method `newDocumentBuilder`. Line 89 begins a `catch` block for a `SAXException`. This exception contains information about errors thrown by the parser.

Figure C.9 presents `MyErrorHandler.java`, which provides the implementation for handling errors thrown by the parser in `ReplaceText.java`. By default, JAXP does not throw any exceptions when a document fails to conform to a DTD. The programmer must provide an error handler, which is registered using method `setErrorHandler` (line 38 in Fig. C.8).

```java
// Fig C.9 : MyErrorHandler.java
// Error Handler for validation errors.

import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class MyErrorHandler implements ErrorHandler {

    // throw SAXException for fatal errors
    public void fatalError( SAXParseException exception )
    throws SAXException {
        throw exception;
    }

    // print any warnings
```

Fig. C.9 Class definition for `MyErrorHandler` (part 1 of 2).
Lines 4–6 of Fig. C.9 import ErrorHandler, SAXException and SAXParseException. Interface ErrorHandler provides methods fatalError, error and warning for fatal errors (i.e., errors that violate the XML 1.0 recommendation; parsing is halted), errors (e.g., validity constraints that do not stop the parsing process) and warnings (i.e., not classified as fatal errors or errors and that do not stop the parsing process), respectively. These methods are overridden in lines 12, 18 and 25. Fatal errors and errors are rethrown and warnings are output to the standard error device (System.err). To run the example, compile the classes and execute class ReplaceText.

Figure C.10 lists the XML document manipulated by the Java application in Fig. C.8. Notice that the message element’s text has been changed and the comments are missing. The DTD is preserved, because we are using a validating parser—otherwise the DTD would not be included in the file. Figure C.10’s output also sets the encoding attribute in the XML declaration. The encoding specifies the character set used in the document. Recall from Appendix A that XML uses the Unicode character set. Unicode provides characters in most of the world’s major languages. Use an encoding (e.g., UTF-8) to specify a subset of the characters in Unicode that will be used in a document.

```java
25     public void warning( SAXParseException warningError )
26     throws SAXParseException
27     {
28         System.err.println( "Warning: " + warningError.getMessage() );
29     }
30 }
```

Fig. C.9  Class definition for MyErrorHandler (part 2 of 2).

```xml
<?xml version = "1.0"?>
<!DOCTYPE myMessage [
<!ELEMENT myMessage ( message )>
<!ELEMENT message ( #PCDATA )>
]>
<myMessage>
<message>Welcome to XML!</message>
</myMessage>
```

Fig. C.10  Input document (intro.xml) and output from ReplaceText.java.
C.5 Creating Nodes

The majority of XML markup presented up to this point has been “hand coded” (i.e., typed into an editor by a document author). Using the DOM, XML documents can be created in an automated way through programming.

Figure C.11 lists a Java application that creates an XML document for a contact list. This application is compiled and executed in the same manner as the last Java application.

```
// Fig. C.11 : BuildXml.java
// Creates element node, attribute node, comment node,
// processing instruction and a CDATA section.

// Java core packages
import java.io.*;

// Java extension packages
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;

// third-party libraries
import org.xml.sax.*;
import org.w3c.dom.*;

public class BuildXml {
    private Document document;

    public BuildXml()
    {
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

        // create new DOM tree
        try {
            DocumentBuilder builder =
                factory.newDocumentBuilder();

            // create root node
            document = builder.newDocument();
        }

        // handle exception thrown by DocumentBuilder
        catch (ParserConfigurationException parserException) {
            parserException.printStackTrace();
        }

        Element root = document.createElement( "root" );
    }

Fig. C.11 Building an XML document with the DOM (part 1 of 3).
```
document.appendChild( root );

// add comment to XML document
Comment simpleComment = document.createComment("This is a simple contact list");
root.appendChild( simpleComment );

// add child element
Node contactNode = createContactNode( document );
root.appendChild( contactNode );

// add processing instruction
ProcessingInstruction pi =
    document.createProcessingInstruction("myInstruction", "action silent");
root.appendChild( pi );

// add CDATA section
CDATASection cdata = document.createCDATASection("I can add <, >, and ?");
root.appendChild( cdata );

// write the XML document to disk
try {
    // create DOMSource for source XML document
    Source xmlSource = new DOMSource( document );
    // create StreamResult for transformation result
    Result result = new StreamResult(
        new FileOutputStream( "myDocument.xml" ) );
    // create TransformerFactory
    TransformerFactory transformerFactory =
        TransformerFactory.newInstance();
    // create Transformer for transformation
    Transformer transformer =
        transformerFactory.newTransformer();
    transformer.setOutputProperty( "indent", "yes" );
    // transform and deliver content to client
    transformer.transform( xmlSource, result );
}

// handle exception creating TransformerFactory
catch (TransformerFactoryConfigurationError factoryError) {
    System.err.println("Error creating " +
        "TransformerFactory");
    factoryError.printStackTrace();
}
Fig. C.11 Building an XML document with the DOM (part 3 of 3).


Lines 44–45 create an element named `root` and append it to the document root. This is the first element appended, so it is the root element of the document. Lines 48–50 create
a comment node using method `createComment` and append the node as a child of element `root`. Line 53 calls programmer-defined method `createContactNode` (line 111) to create the `contact` element. We will discuss this method momentarily. Lines 57–58 create a processing instruction node. The first argument passed to `createProcessingInstruction` is the target `myInstruction`, and the second argument passed is the value `action silent`. Line 60 appends the processing instruction to the root element. Lines 63–64 create a `CDATA` section, which is appended to element `root` in line 65.

Line 111 defines method `createContactNode` that returns a `Node` object. This method creates a `contact` element node and returns it. The returned `Node` is appended to the `root` element in line 53. Lines 115–116 create elements `FirstName` and `LastName`, which have their text values set on lines 118–119. Lines 125–126 create attribute `gender` using method `createAttribute` and assign it a value using `Attr` method `setValue`. Line 129 assigns the attribute to the `contact` element node using method `setAttributeNode`.

The XML document is written to disk in lines 68–89. Figure C.12 lists the XML document (`myDocument.xml`) created by `BuildXml.java` (Fig. C.11).

**C.6 Traversing the DOM**

In this section, we demonstrate how to use the DOM to traverse an XML document. In Fig. C.13, we present a Java application that outputs element nodes, attribute nodes and text nodes. This application takes the name of an XML document (e.g., `simpleContact.xml` in Fig. C.14) as an argument.

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<root>
<!--This is a simple contact list-->
<contact gender = "F">
 <FirstName>Sue</FirstName>
 <LastName>Green</LastName>
</contact>
<?myInstruction action silent?>
<![CDATA[I can add <, >, and ?]]>
</root>
```

Fig. C.12 Output for `buildXml.java`.

```java
// Fig. C.13 : TraverseDOM.java
// Traverses DOM and prints various nodes.

// Java core packages
import java.io.*;

// Java extension packages
import javax.xml.parsers.*;

// third-party libraries
```

Fig. C.13 Traversing the DOM (part 1 of 3).
```java
import org.w3c.dom.*;
import org.xml.sax.*;

public class TraverseDOM {
    private Document document;

    public TraverseDOM( String file )
    {
        // parse XML, create DOM tree, call method processNode
        try {

            // obtain default parser
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            factory.setValidating( true );
            DocumentBuilder builder = factory.newDocumentBuilder();

            // set error handler for validation errors
            builder.setErrorHandler( new MyErrorHandler() );

            // obtain document object from XML document
            document = builder.parse( new File( file ) );
            processNode( document );
        }

        // handle exception thrown by DocumentBuilder
        catch ( ParserConfigurationException parserException ) {
            parserException.printStackTrace();
        }

        // handle exception thrown by Parser
        catch ( SAXException saxException ) {
            saxException.printStackTrace();
        }

        // handle exception thrown when reading data from file
        catch ( IOException ioException ) {
            ioException.printStackTrace();
            System.exit( 1 );
        }
    }

    public void processNode( Node currentNode )
    {
        switch ( currentNode.getNodeType() ) {

            // process Document root
            case Node.DOCUMENT_NODE:
                Document doc = ( Document ) currentNode;
                System.out.println(
                    "Document node: " + doc.getNodeName() +
                    "\nRoot element: " +
                    ( doc.getChildNodes() == null ? "null" : doc.getFirstChild().getNodeName() ) +
                );

        }
    }
}
```

Fig. C.13 Traversing the DOM (part 2 of 3).
```java
doc.getDocumentElement().getNodeName() );
processChildNodes( doc.getChildNodes() );
break;

// process Element node
case Node.ELEMENT_NODE:
    System.out.println( "\nElement node: " +
            currentNode.getNodeName() );
    NamedNodeMap attributeNodes =
        currentNode.getAttributes();
    for ( int i = 0; i < attributeNodes.getLength(); i++) {
        Attr attribute = ( Attr ) attributeNodes.item( i );
        System.out.println( "\tAttribute: " +
            attribute.getNodeName() + " ; Value = " +
            attribute.getNodeValue() );
    }
    processChildNodes( currentNode.getChildNodes() );
break;

// process text node and CDATA section
case Node.CDATA_SECTION_NODE:
    case Node.TEXT_NODE:
        Text text = ( Text ) currentNode;
        if ( !text.getNodeValue().trim().equals( "" ) )
            System.out.println( "\nText: " +
                text.getNodeValue() );
        break;
    }

public void processChildNodes( NodeList children )
{
    if ( children.getLength() != 0 )
        for ( int i = 0; i < children.getLength(); i++ )
            processNode( children.item( i ) );
}

public static void main( String args[] )
{
    if ( args.length < 1 ) {
        System.err.println( "Usage: java TraverseDOM <filename>" );
        System.exit( 1 );
    }
    TraverseDOM traverseDOM = new TraverseDOM( args[ 0 ] );
}
```

Fig. C.13  Traversing the DOM (part 3 of 3).
Lines 17–51 define the class constructor for TraverseDOM, which takes the name of the file (i.e., simpleContact.xml) specified at the command line, loads and parses the XML document before passing it to programmer-defined method processNode.

Lines 53–96 define method processNode, which takes one Node argument and outputs information about the Node and its child elements. Line 55 begins a switch structure that determines the Node type.

Line 58 matches the document node. This case outputs the document node (represented as #document) and processes its child nodes by calling method processChildNodes (lines 98–104). We will discuss method processChildNodes momentarily. Line 69 matches an element node. This case outputs the element’s attributes and then processes its child nodes in processChildNodes.

Lines 87–88 match CDATA section nodes and text nodes. These cases output the node’s text content (lines 91–93).

Lines 98–104 define method processChildNodes, which takes one NodeList argument and calls processNode on a node’s child nodes. Each child node is retrieved by calling NodeList method item (line 103).

Figure C.14 lists the contents of simpleContact.xml, the XML document used by TraverseDOM.java.

### C.7 Internet and World Wide Web Resources

[www.w3.org/DOM](http://www.w3.org/DOM)
W3C DOM home page.

[www.w3schools.com/dom/default.asp](http://www.w3schools.com/dom/default.asp)
The W3Schools DOM introduction, tutorial and links site.

[www.oasis-open.org/cover/dom.html](http://www.oasis-open.org/cover/dom.html)
The Oasis-Open DOM page contains a comprehensive overview of the Document Object Model with references and links.

[dmoz.org/Computers/Programming/Internet/W3C_DOM](http://dmoz.org/Computers/Programming/Internet/W3C_DOM)
This is a useful set of DOM links to different locations and instructional matter.

[www.w3.org/DOM/faq.html](http://www.w3.org/DOM/faq.html)
Answers to DOM Frequently Asked Questions (FAQs).

[www.jdom.org](http://www.jdom.org)
Home page for the JDOM XML API in Java.

---

```xml
1  <?xml version = "1.0"?>
2  
3  <!-- Fig C.14 : simpleContact.xml -->
4  <!-- Input file for traverseDOM.java -->
5  
6  <!DOCTYPE contacts [ 
7  <!ELEMENT contacts ( contact+ )>
8  <!ELEMENT contact ( FirstName, LastName )>
9  <!ATTLIST contact gender ( M | F ) "M"> 
10  <!ELEMENT FirstName ( #PCDATA )>
11  <!ELEMENT LastName ( #PCDATA )>
12  ]>
```

Fig. C.14 Sample execution of TraverseDOM.java (part 1 of 2).
SUMMARY

- XML documents, when parsed, are represented as a hierarchical tree structure in memory. This tree structure contains the document’s elements, attributes, text, etc. XML was designed to be a live, dynamic technology—the contents of the tree structure can be modified by a programmer. This essentially allows the programmer to add data, remove data, query for data, etc., in a manner similar to manipulating a database.

- The W3C provides a recommendation for building a tree structure in memory for XML documents called the XML Document Object Model (DOM). Any parser that adheres to this recommendation is called a DOM-based parser.

- A DOM-based parser exposes (i.e., makes available) a programmatic library—called the DOM Application Programming Interface (API)—that allows data in an XML document to be accessed and manipulated. This API is available for many different programming languages.

- Property `documentElement` returns a document’s root element node. The root element node is a reference point for retrieving child elements, text, etc.

- Property `nodeName` returns the name of an attribute, element, etc., which are collectively called nodes.

- Property `childNodes` contains a node’s child nodes. Property `length` returns the number of child nodes.

- Individual child nodes are accessed using the `item` method. Each node is given an integer value (starting at zero) based on the order in which the nodes occur in the XML document.

- Property `firstChild` retrieves the root node’s first child node.
Nodes at the same level in a document (i.e., that have the same parent node) are called siblings. Property `nextSibling` returns a node’s next sibling.

A text node’s value is its text, an element node’s value is `null` (which indicates the absence of a value) and an attribute node’s value is the attribute’s value.

Property `parentNode` returns a node’s parent node.

The `Document` object represents the top-level node of an XML document in memory and provides a means of creating nodes and retrieving nodes.

Interface `Node` represents an XML document node.

`Element` represents an element node.

Sun Microsystems provides several packages related to XML. Package `org.w3c.dom` provides the DOM-API programmatic interface (i.e., classes, methods, etc.). Package `javax.xml.parsers` provides classes related to parsing an XML document. Classes in `javax.xml.transform.*` are necessary to transform and output XML documents.

A DOM-based parser may use an event-based implementation (i.e., as the document is parsed, events are raised when starting tags, attributes, etc., are encountered) to help create the tree structure in memory. A popular event-based implementation is called the Simple API for XML (SAX). Package `org.xml.sax` provides the SAX programmatic interface.

Class `DocumentBuilderFactory` (package `javax.xml.parsers`) obtains an instance of a parser.

Method `setValidating` specifies whether a parser is validating or nonvalidating.

Method `parse` loads and parses XML documents. If parsing is successful, a `Document` object is returned. Otherwise, a `SAXException` is thrown.

Method `getDocumentElement` returns the `Document`’s root node. The `Document`’s root node represents the entire document—not the root element node.

Method `getNodeType` retrieves the node’s type.

Elements in the XML document are retrieved by calling method `getElementsByTagName`. Each element is stored as an item (i.e., a `Node`) in a `NodeList`. The first item added is stored at index 0, the next at index 1, and so forth. This index is used to access an individual item in the `NodeList`.

Interface `Text` represents an element’s or attribute’s character data.

Method `replaceChild` replaces a `Node`.

`SAXParseException` and `SAXException` contain information about errors and warnings thrown by the parser. Class `SAXParseException` is a subclass of `SAXException` and includes methods for locating the error.

By default, JAXP does not throw any exceptions when a document fails to conform to a DTD. Programmers must provide their own implementation, which is registered using method `setErrorHandling`.

Interface `ErrorHandler` provides methods `fatalError`, `error` and `warning` for fatal errors (i.e., errors that violate the XML 1.0 syntax; parsing is halted), errors (e.g., such as validity constraints that do not stop the parsing process) and warnings (i.e., not classified as fatal errors or errors and that do not stop the parsing process), respectively.

Method `newDocument` creates a new `Document` object, which can be used to build an XML document in memory.

Method `createComment` creates a comment node.

Method `createProcessingInstruction` creates a processing instruction, and method `createCDATASection` creates a `CDATA` section node.
**TERMINOLOGY**

API  
appendChild  
Attr  
CDATA section  
childNodes  
cloneNode  
Comment  
comment node  
createAttribute  
createCDATASection  
createComment  
createElement  
createProcessingInstruction  
createTextNode  
createXmlDocument  
Document  
document root  
DocumentBuilder  
DocumentBuilderFactory  
documentElement  
DOM  
Element  
encoding  
declaration  
firstChild  
getAttribute  
getChildNodes  
getDocumentElement  
getNodeName  
getNodeType  
getNodeValue  
getParentNode  
getTagName  
hasChildNodes  
item  
method  
Java API for XML Parsing (JAXP)  
length  
nextSibling  
Node  
Node.ATTRIBUTE_NODE  
Node.CDATA_SECTION_NODE  
Node.COMMENT_NODE  
Node.ELEMENT_NODE  
Node.PROCESSING_INSTRUCTION_NODE  
Node.TEXT_NODE  
nodeList  
nodeName  
noduleValue  
parentNode  
processing instruction node  
ProcessingInstruction  
removeAttribute  
removeChild  
replaceChild  
root  
element node  
setAttributes  
setNodeValue  
setOutputProperty  
setValue  
method of Attr  
Text  
Transformer  
TransformerFactory  
write

**SELF-REVIEW EXERCISES**

**C.1** State whether each of the following are true or false. If false, explain why.

a) Class Transformer is a class in the JAXP API.
b) Text in XML document is not represented as a node.
c) A NodeList contains a list of Nodes.
d) Interface CDATASection extends interface CharacterData.
e) Interface Attr extends interface Element.
f) Method parse loads and parses an XML document.
g) Interface ErrorHandler defines methods fatalError, error and warning.
h) Method getElementByTagName of class Element returns the first element in the XML document that matches the specified name.
i) The replaceChild method of Node can be used to replace child node with another node.
j) An element’s node value is text (i.e., character data).

**C.2** Fill in the blanks in each of the following statements:

a) DOM is an acronym for ________.
b) DOM-based parsers represent an XML document’s data as a _________ structure.
c) The number of Nodes in a NodeList is determined by calling method _________.
d) Method ________ replaces one child node with another.
e) Method ________ creates a text node.
f) Method ________ returns a NodeList containing every occurrence of a particular element.
g) ________ represents the root of an XML document.
h) ________ is a static Node constant that represents an element.

ANSWERS TO SELF-REVIEW EXERCISES
C.1  a) True.  b) False. Text is represented as a node. c) True.  d) True.  e) False. Interface Attr is derived from interface Node. f) True.  g) True.  h) False. Method getElementByTagName returns a NodeList containing all the element nodes that match the specified name. i) True.  j) False. An element’s node value is null.
C.2  a) Document Object Model. b) tree. c) getLength. d) replaceChild. e) createTextNode. f) getElementsByTagName. g) Document. h) NODE.ELEMENT_NODE.
XSL: Extensible Stylesheet Language Transformations (XSLT)

Objectives

- To understand what the Extensible Stylesheet Language is and how it relates to XML.
- To understand what an Extensible Stylesheet Language Transformation (XSLT) is.
- To be able to write XSLT documents.
- To be able to write templates.
- To be able to iterate through a node set.
- To be able to sort.
- To be able to perform conditional processing.
- To be able to declare variables.

*Guess if you can, choose if you dare.*
Pierre Corneille

*A Mighty Maze! but not without a plan.*
Alexander Pope

*Behind the outside pattern*
the dim shapes get clearer every day.
*It is always the same shape, only very numerous.*
Charlotte Perkins Gilman
D.1 Introduction

The Extensible Stylesheet Language (XSL) provides rules for formatting XML documents. In this appendix, we present the XSL Transformation Language (XSLT). XSLT transforms an XML document to another text-based form, such as XHTML. We present many XSLT examples and show the results of the transformations.

D.2 Applying XSLTs with Java

To process XSLT documents, you will need an XSLT processor. We have created a Java program that uses the XSLT processor in the JAXP library to perform XSL transformations. Our program, Transform.java (Fig. D.1), takes as command-line arguments the name of the XML document to be transformed, the name of the XSLT document and the name of the document that will be created by the transformation.

```java
// Fig. D.1: Transform.java
// Performs XSL Transformations.

// Java core libraries
import java.io.*;
import java.util.*;

// Java standard extensions
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// third-party libraries
import org.w3c.dom.*;
import org.xml.sax.SAXException;
```

Fig. D.1 Java application that performs XSL transformations (part 1 of 2). © Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 8/23/01
public class Transform {

    // execute application
    public static void main( String args[] ) throws Exception {
        if ( args.length != 3 ) {
            System.err.println( "Usage: java Transform input.xml"
                + " input.xsl output.xml" );
            System.exit( 1 );
        }

        // factory for creating DocumentBuilders
        DocumentBuilderFactory builderFactory =
            DocumentBuilderFactory.newInstance();

        // factory for creating Transformers
        TransformerFactory transformerFactory =
            TransformerFactory.newInstance();

        DocumentBuilder builder =
            builderFactory.newDocumentBuilder();

        Document document = builder.parse( new File( args[ 0 ] ) );

        // create DOMSource for source XML document
        Source xmlSource = new DOMSource( document );

        // create StreamSource for XSLT document
        Source xslSource = new StreamSource( new File( args[ 1 ] ) );

        // create StreamResult for transformation result
        Result result = new StreamResult( new File( args[ 2 ] ) );

        // create Transformer for XSL transformation
        Transformer transformer =
            transformerFactory.newTransformer( xslSource );

        // transform and deliver content to client
        transformer.transform( xmlSource, result );
    }
}

Fig. D.1 Java application that performs XSL transformations (part 2 of 2).

Lines 1–38 import the necessary classes and create a DocumentBuilderFactory, a DocumentBuilder and a TransformerFactory. Line 40 creates a Document object, passing it the XML file to be transformed. Line 43 creates a new DOMSource object from which the Transformer reads the XML Document. Line 46 creates a StreamSource object from which the Transformer reads the XSL File. DOMSource and StreamSource both implement the Source interface. Line 49 creates a StreamResult object to which the Transformer writes a File containing the result of the transformation. Lines 52–53 call method newTransformer, passing it the Source object of the XSL file as an argument. Constructing a Transformer with an
XSL file allows the Transformer to apply the rules in that file when transforming XML documents. Line 56 applies the XSL transformation to the XML file to produce the result File. In this appendix, you will use this Java application to apply XSL transformations to XML files.

[Note: In this example we use a DOMSource to provide the XML document to the Transformer, although we could have used a StreamSource to read from the XML file directly. We chose this method, because in other examples in the book, we build the DOM tree in memory and do not have a file from which to create a StreamSource object.]

### D.3 Templates

An XSLT document is an XML document with a root element stylesheet. The namespace for an XSLT document is http://www.w3.org/1999/XSL/Transform. The XSLT document shown in Fig. D.2 transforms intro.xml (Fig. D.3) into a simple XHTML document (Fig. D.4).

XSLT uses XPath expressions to locate nodes in an XML document. [Note: A structured complete explanation of the XPath language is beyond the scope of this book. We explain each XPath expression as we encounter it in the examples.] In an XSL transformation, there are two trees of nodes. The first node tree is the source tree. The nodes in this tree correspond to the original XML document to which the transformation is applied. The second node tree is the result tree. The result tree contains all of the nodes produced by the XSL transformation. This result tree represents the document produced by the transformation. The document used as the source tree is not modified by an XSL transformation.

Lines 6–7 contain the XSLT document’s root element (i.e., xsl:stylesheet). Attribute version indicates the XSLT specification used. Namespace prefix xsl is defined and assigned the URI "http://www.w3.org/1999/XSL/Transform".

Line 9 contains a template element. This element matches specific XML document nodes by using an XPath expression in attribute match. In this case, we match any myMessage element nodes.

```xml
<?xml version = "1.0"?>
<!-- Fig. D.2 : intro.xsl               -->
<!-- Simple XSLT document for intro.xml -->
<xsl:stylesheet version = "1.0"
xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:template match = "myMessage">
<html>
<body><xsl:value-of select = "message"/></body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Fig. D.2 Simple template.
Lines 10–12 are the contents of the template element. When a myMessage element node is matched in the source tree (i.e., the document being transformed), the contents of the template element are placed in the result tree (i.e., the document created by the transformation). By using element value-of and an XPath expression in attribute select, the text contents of the node(s) returned by the XPath expression are placed in the result tree.

Figure D.3 lists the input XML document. Figure D.4 lists the results of the transformation.

### D.4 Creating Elements and Attributes

In the previous section, we demonstrated the use of XSLT for simple element matching. This section discusses the creation of new elements and attributes within an XSLT document. Figure D.5 lists an XML document that marks up various sports.

Figure D.6 lists the XSLT document that transforms the XML document in Fig. D.5 into another XML document.

```
<?xml version = "1.0"?>
<!-- Fig. D.3 : intro.xml              -->
<!-- Simple introduction to XML markup -->
<myMessage>
  <message>Welcome to XSLT!</message>
</myMessage>
```

**Fig. D.3** Sample input XML document *intro.xml*.

```
<html>
<body>Welcome to XSLT!</body>
</html>
```

**Fig. D.4** Results of XSL transformation.
<?xml version = "1.0"?>
<!-- Fig. D.5 : games.xml -->
<!-- Sports Database -->

<games>
  <game title = "cricket">
    <id>243</id>
    <para>
      More popular among commonwealth nations.
    </para>
  </game>
  <game title = "baseball">
    <id>431</id>
    <para>
      More popular in America.
    </para>
  </game>
  <game title = "soccer">
    <id>123</id>
    <para>
      Most popular sport in the world.
    </para>
  </game>
</games>

Fig. D.5 XML document containing a list of sports.

<?xml version = "1.0"?>
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = "/">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match = "sports">
    <sports>
      <xsl:apply-templates/>
    </sports>
  </xsl:template>
</xsl:stylesheet>

Fig. D.6 Using XSLT to create elements and attributes (part 1 of 2).
Lines 9–11 use the **match** attribute to select the document root (i.e., the parent of the root element) of the XML document. The `/` character represents the document root. Element **apply-templates** applies the **templates** to specific nodes. In this case, we have not specified any particular nodes. By default, element **apply-templates** matches all the child nodes of an element. In Fig. D.5, the child nodes of the document root are two comment nodes and the **sports** element node.

The XSLT recommendation defines **default templates** for the nodes of an XML document. If a programmer does not specify a **template** that matches a particular element, the default XSLT template is applied. Default templates are described in Fig. D.7.

Lines 13–17 match element **sports**. We output the **sports** element and apply templates to the child nodes of the **sports** element. Line 19 matches element **game**. In the input XML document, element **game** contains the name of a sport, its unique identifier and a description.

---

**Template / Description**

```xml
<xsl:template match = "/*" |
  <xsl:apply-templates/>
</xsl:template>
This template matches and applies templates to the child nodes of the document root (/) and any element nodes (*).

<xsl:template match = "text() | @*" |
  <xsl:value-of select = "."/>
</xsl:template>
This template matches and outputs the values of text nodes (text()) and attribute nodes (@).
```

---

**Fig. D.6** Using XSLT to create elements and attributes (part 2 of 2).
Line 20 shows the element `element`, which creates an element, with the element name specified in attribute `name`. Therefore, the name of this XML element will be the name of the sport contained in attribute `title` of element `game`. [Note: In XPath, the symbol `@` specifies an attribute.]

Lines 22–24 show element `attribute`, which creates an attribute for an element. Attribute `name` provides the name of the attribute. The text in element `attribute` specifies the attribute’s value in the result tree. Here, the attribute statement will create attribute `id` for the new element, which contains the text in element `id` of element `game`. Lines 26–28 create element `comment` with the contents of element `para`.

Figure D.8 lists the output of the transformation. Your output may not look exactly like that in the figure, because we have modified the output in the figure for presentation. The original XML document has been transformed into a new XML document with sport names as elements (instead of attributes, as in the original document).

### D.5 Iteration and Sorting

XSLT allows for iteration through a node set (i.e., all nodes an XPath expression matches). Node sets can also be sorted in XSLT. Figure D.9 shows an XML document containing information about a book.

```xml
1  <?xml version = "1.0" encoding = "UTF-8"?>
2  <sports>
3
4      <cricket id = "243">
5          <comment>
6              More popular among commonwealth nations.
7          </comment>
8      </cricket>
9
10     <baseball id = "431">
11          <comment>
12              More popular in America.
13          </comment>
14     </baseball>
15
16     <soccer id = "123">
17          <comment>
18              Most popular sport in the world.
19          </comment>
20 </soccer>
```

Fig. D.8 Output of transformation (part 1 of 2).
Figure D.10 shows the XSLT style sheet that transforms the document in Fig. D.9.

```
<?xml version = "1.0"?>
<!-- Fig. D.9 : usage.xml -->
<!-- Book information -->
<book isbn = "999-99999-9-X"
    title="Deitel's XML Primer"
    author="Paul Deitel"
    chapters="Welcome, XML Elements?, Easy XML, Entities"
    media type="CD"/>
```

Figure D.10 shows the XSLT style sheet that transforms the document in Fig. D.9.

```
<?xml version = "1.0"?>
<!-- Fig. D.10 : usage.xsl -->
<!-- Transformation of book information into XHTML -->
<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
    xml:stylesheet="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match = "/"
      html>
    <head>
      <xsl:value-of select = "@title"/>
    </head>
    <xsl:apply-templates/>
  </html>
</xsl:template>
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 8/23/01
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="xsl:stylesheet">
    <body>
      <h1><xsl:value-of select = "title"/></h1>
      <h2>by <xsl:value-of select = "author/lastName"/>
        , \n        <xsl:value-of select = "author/firstName"/></h2>

      <table border = "1">
        <xsl:for-each select = "chapters/preface">
          <xsl:sort select = "@num" order = "ascending"/>
          <tr>
            <td align = "right">
              Preface <xsl:value-of select = "@num"/>
            </td>
            <td>
              <xsl:value-of select = "."/> ( 
                <xsl:value-of select = "@pages"/> pages )
            </td>
          </tr>
        </xsl:for-each>

        <xsl:for-each select = "chapters/chapter">
          <xsl:sort select = "@num" order = "ascending"/>
          <tr>
            <td align = "right">
              Chapter <xsl:value-of select = "@num"/>
            </td>
            <td>
              <xsl:value-of select = "."/> ( 
                <xsl:value-of select = "@pages"/> pages )
            </td>
          </tr>
        </xsl:for-each>

        <xsl:for-each select = "chapters/appendix">
          <xsl:sort select = "@num" order = "ascending"/>
          <tr>
            <td align = "right">
              Appendix <xsl:value-of select = "@num"/>
            </td>
            <td>
              <xsl:value-of select = "."/> ( 
                <xsl:value-of select = "@pages"/> pages )
            </td>
          </tr>
        </xsl:for-each>

      </table>
    </body>
  </xsl:template>
</xsl:stylesheet>

Fig. D.10 Transforming XML data into XHTML (part 2 of 2).
Line 15 is an XSLT template that matches the book element. In this template, we construct the body of the XHTML document. Lines 17–18 create the title for the XHTML document. We use the ISBN of the book from attribute isbn and also the contents of element title to create the title string, resulting in ISBN 999-99999-9-X - Deitel's XML Primer. Line 22 creates a header element with the title of the book, selected from element title. Lines 24–25 create another header element, displaying the name of the book’s author. The XPath expression author/lastName selects the author’s last name, and the expression author/firstName selects the author’s first name.

Line 28 shows XSLT element for-each, which applies the contents of the element to each of the nodes selected by attribute select. In this case, we select all preface elements of the chapters element. XSLT element sort (line 29) sorts the nodes selected by the for-each element by the field in attribute select. Attribute order specifies how these nodes should be ordered. Attribute order has possible values such as "ascending" (i.e., A–Z) and "descending" (i.e., Z–A). In this for-each element, we sort the nodes by attribute num, in ascending order.

Lines 30–39 output a table row displaying the preface number, the title of the preface and the number of pages in that preface for each preface element. Similarly, lines 42–54 output the chapter elements, and lines 56–68 output the appendix elements.

Figure D.11 shows the results of the transformation. Your output may look different, because we have modified ours for presentation. Notice that the chapters have appeared in the correct order, even though the XML document contained the elements in a different order.

```html
<html>
<head>
</head>
<body>
<h1>Deitel's XML Primer</h1>
<h2>by Deitel, Paul</h2>
<table border = "1">
<tr>
<td align = "right">Preface 1</td>
<td>Welcome ( 2 pages )</td>
</tr>
<tr>
<td align = "right">Chapter 1</td>
<td>Easy XML ( 4 pages )</td>
</tr>
<tr>
<td align = "right">Chapter 2</td>
<td>XML Elements? ( 2 pages )</td>
</tr>
<tr>
<td align = "right">Appendix 1</td>
</tr>
</table>
</body>
</html>
```

Fig. D.11  Output of the transformation (part 1 of 2).
D.6 Conditional Processing

In the previous section, we discussed the iteration of a node set. XSLT also provides elements to perform conditional processing, such as `if` elements. Figure D.12 is an XML document that a day planner application might use.

```xml
<?xml version = "1.0"?>
<!-- Fig. D.12 : planner.xml -->
<!-- Day Planner XML document -->

<planner>
  <year value = "2001">
    <date month = "7" day = "15">
      <note time = "1430">Doctor's appointment</note>
      <note time = "1620">Physics class at BH291C</note>
    </date>
    <date month = "7" day = "4">
      <note>Independence Day</note>
    </date>
  </year>
</planner>
```

Fig. D.12 Day planner XML document (part 1 of 2).

Fig. D.11 Output of the transformation (part 2 of 2).
Figure D.13 is an XSLT document for transforming the day planner XML document into an XHTML document.

XSLT provides the `choose` element (lines 34–60) to allow alternative conditional statements, similar to a nested `if/else` structure in Java. Element `when` corresponds to an `if` statement. The `test` attribute of the `when` element specifies the expression that is being tested. If the expression is true, the XSLT processor evaluates the code inside the `when` element. Line 37, for instance, is an expression that will be true when the `time` is between "0500" and "1200". The element `choose` serves to group all the `when` elements, thereby making them exclusive of one another (i.e., the first `when` element whose conditional is satisfied will be executed). The element `otherwise` (lines 56–58) corresponds to the final `else` statement in a nested `if/else` structure and is likewise optional.

Lines 64–66 show the `if` conditional statement. This `if` determines whether the current node (represented as `.`) is empty. If so, `n/a` is inserted into the result tree. Unlike element `choose`, element `if` provides a single conditional test.
Fig. D.13 Using conditional elements (part 2 of 3).
D.7 Combining Style Sheets

XSLT allows for modularity in style sheets. This feature enables XSLT documents to import other XSLT documents. Figure D.14 lists an XSLT document that is imported into the XSLT document in Fig. D.15 using element `import`.

```
<?xml version = "1.0"?>
<!-- Fig. D.14 : usage2.xsl -->
<!-- xsl:import example -->
<xsl:stylesheet version = "1.0"
                   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
<xsl:template match = "book">
  <html>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Fig. D.14 XSLT document being imported (part 1 of 2).
The XSLT document in Fig. D.14 is similar to the ones presented earlier. One notable difference is line 33, which provides a template to match any element or text nodes. When these nodes are matched, this template indicates that no data will be written to the result tree. If the template is not provided, default templates will output the other nodes.

Line 9 in Fig. D.15 uses element import to use the templates defined in the XSLT document (Fig. D.14) referenced by attribute href.

Line 13 provides a template for element title, which already has been defined in the XSLT document being imported. This local template has higher precedence than the imported template, so it is used instead of the imported template. Figure D.16 shows the result of transforming usage.xml (Fig. D.9).

---

<?xml version = "1.0"?>
<!-- Fig. D.15 : usage1.xsl -->
<! xsl:import example using usage.xml -->
<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
    <xsl:import href = "usage2.xsl"/>
    <!-- this template has higher precedence over templates being imported -->
    <xsl:template match = "title">
        <h2>
            <xsl:value-of select = "."/>
        </h2>
    </xsl:template>
</xsl:stylesheet>

---
Figure D.17 shows an example of the XSLT element *include*, which includes other XSLT documents in the current XSLT document. Lines 28–29 contain element *include*, which includes the files referenced by attribute *href*. The difference between element *include* and element *import* is that templates that are included have the same precedence as the local templates. Therefore, if any templates are duplicated, the template that occurs last is used.
Figure D.17  Combining style sheets using \texttt{xsl:include} (part 2 of 2).

Figure D.18 and Figure D.19 list the XSLT documents being included by Figure D.17.

Fig. D.18  XSLT document for rendering the author’s name.

<?xml version = "1.0"?>
<!-- Fig. D.18 : author.xsl -->
<xs1:stylesheet version = "1.0"
xmns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xs1:template match = "author">
    <p>Author:
      <xs1:value-of select = "lastName" />, 
      <xs1:value-of select = "firstName" />
    </p>
  </xs1:template>
</xs1:stylesheet>

Fig. D.19  XSLT document for rendering chapter names (part 1 of 2).

<?xml version = "1.0"?>
<!-- Fig. D.19 : chapters.xsl -->
<xs1:stylesheet version = "1.0"
xmns:xsl = "http://www.w3.org/1999/XSL/Transform">
  <xs1:template match = "chapters">
    Chapters:
    <ul>
    <xs1:apply-templates select = "chapter"/>
    </ul>
  </xs1:template>
</xs1:stylesheet>
Figure D.20 shows the result of the XSLT document (Fig. D.17) applied to the XML document describing a book (Fig. D.9).

```xml
<xsl:template match = "author">
  <p><strong>Author is: </strong>
    <xsl:value-of select = "lastName"/>
    ,
    <xsl:value-of select = "firstName"/>
  </p>
</xsl:template>

<xsl:template match = "chapter">
  <li>
    <xsl:value-of select = "."/>
  </li>
</xsl:template>
```

Fig. D.19  XSLT document for rendering chapter names (part 2 of 2).

Figure D.20 shows the result of the XSLT document (Fig. D.17) applied to the XML document describing a book (Fig. D.9).

```html
<html>
<body>
<h2>Deitel's XML Primer</h2>
  <strong>Author is: </strong>Deitel, Paul
  <h2>Chapters:</h2>
  <ul>
    <li>XML Elements?</li>
    <li>Easy XML</li>
  </ul>
</body>
</html>
```

Fig. D.20  Output of an XSLT document using element include.
D.8 Variables

XSLT provides variables for storing information. These variables are not like Java variables, but rather more like Java constants. Figure D.21 provides an example of an XSLT document using element `variable`.

Lines 13–14 use element `variable` to create a variable named `pageNumber`, which stores the total number of pages in the book. Attribute `select` has the value "sum(book/chapters/*/@pages)", which is an XPath expression for summing the number of `pages`. This XPath expression uses function `sum` to iterate over a set of nodes and sum their values. The set of nodes includes any element (*) containing an attribute `pages` that is a child of `book/chapters`.

Line 15 uses element `value-of` to output the variable `pageNumber`’s value. The dollar sign ($) references the variable’s content.

Figure D.22 shows the output of the XSLT document (Fig. D.21) when it is applied to `usage.xml` (Fig. D.9). The total number of pages is 17.

D.9 Internet and World Wide Web Resources

`www.w3.org/Style/XSL`
The W3C Extensible Stylesheet Language Web site.

```
<?xml version = "1.0"?>
<total>
  Number of pages =
  <xsl:variable name = "pageNumber"
    select = "sum(book/chapters/*/@pages)"/>
  <xsl:value-of select = "pageNumber"/>
</total>
</xsl:stylesheet>
```

Fig. D.21  Demonstrating `xsl:variable`.

```
<?xml version="1.0" encoding="UTF-8"?><total>
  Number of pages =
  17</total>
```

Fig. D.22  Result of `variables.xsl` transformation.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 8/23/01
www.w3.org/TR/xsl
The most current W3C XSL recommendation.

www.w3schools.com/xsl
This site features an XSL tutorial, along with a list of links and resources.

www.dpawson.co.uk/xsl/xslfaq.html
A comprehensive collection of XSL FAQs.

www.bayes.co.uk/xml/index.xml
A portal site that heavily uses XML and XSL.

msdn.microsoft.com/xml
Microsoft Developer Network XML home page, which provides information on XML and XML-related technologies, such as XSL/XSLT.

xml.apache.org/xalan-j/index.html
Home page for Apache's XSLT processor Xalan.

java.sun.com/xml/xml_jaxp.html
Home page for JAXP, an implementation of XSLT in Java.

**SUMMARY**

- XSL Transformation Language (XSLT) transforms XML documents into other text-based documents using XSL format instructions. XSLT uses XPath to match nodes when transforming an XML document into a different document. The resulting document may be XML, XHTML, plain text or any other text-based document.

- To process XSLT documents, you will need an XSLT processor such as the **Transformer**-related classes in JAXP.

- An XSLT document is an XML document with a root element **xsl:stylesheet**. Attribute **version** indicates the XSLT specification used.

- An XSLT document’s namespace URI is [http://www.w3.org/1999/XSL/Transform](http://www.w3.org/1999/XSL/Transform).

- A **template** element matches specific XML document nodes by using an XPath expression in attribute **match**.

- Element **apply-templates** applies an XSLT document’s templates to specific element nodes. By default, element **apply-templates** matches all element child nodes.

- The XSLT specification defines default templates for an XML document’s nodes. The template

  ```xml
  <xsl:template match = "/ | *">
    <xsl:apply-templates/>
  </xsl:template>
  ```

  matches the document root node and any element nodes of an XML document and applies templates to their child nodes. The template

  ```xml
  <xsl:template match = "text() | @*">
    <xsl:value-of select = "."/>
  </xsl:template>
  ```

  matches text nodes and attribute nodes and outputs their values. The template

  ```xml
  <xsl:template match = "processing-instruction() | comment()"/>
  ```

  matches processing-instruction nodes and comment nodes, but does not perform any actions with them.
• Element **element** creates an element with the name specified in attribute **name**.

• Element **attribute** creates an attribute for an element and can be contained only within an **element** element. Attribute **name** provides the name of the attribute.

• XSLT provides the capability to iterate through a node set returned by an XPath expression. XSLT also provides the capability to sort a node set.

• XSLT element **for-each** applies the element’s contents to each of the nodes specified by attribute **select**.

• Element **sort** sorts (in the order specified by attribute **order**) the nodes specified in attribute **select**. Attribute **order** has values **ascending** (i.e., A–Z) and **descending** (i.e., Z–A).

• XSLT provides elements to perform conditional processing.

• Element **choose** allows alternative conditional statements to be processed.

• XSLT allows for modularity in style sheets. This feature allows XSLT documents to import other XSLT documents by using element **import**. Other XSLT document are referenced using attribute **href**.

• Local templates have higher precedence than imported templates. XSLT element **include** includes other XSLT documents in the current XSLT document.

• The difference between element **include** and element **import** is that templates included using element **include** have the same precedence as the local templates. Therefore, if any templates are duplicated, the template that occurs last is used.

• XSLT provides variables for storing values.

**TERMINOLOGY**

- **$**
- **ascending**
- **attribute href**
- **attribute match**
- **attribute name**
- **attribute order**
- **attribute select**
- **attribute test**
- **attribute type**
- **attribute version**
- **choose element**
- **conditional processing**
- **descending**
- **DOMSource**
- **element apply-templates**
- **element attribute**
- **element element**
- **element for-each**
- **element if**
- **element import**
- **element include**
- **element otherwise**
- **element sort**
- **element value-of**
- **element variable**
- **element when**
- **Extensible Stylesheet Language (XSL)**
- **http://www.w3.org/1999/XSL/Transform URI**
- **if conditional statement**
- **otherwise conditional**
- **root element stylesheet**
- **StreamSource**
- **template element**
- **text-based document**
- **when conditional**
- **XPath**
- **XPath expression**
- **XSL (XSL Transformation Language)**
- **XSLT processor**

**SELF-REVIEW EXERCISES**

D.1 State whether the following are true or false. If false, explain why.
a) XSLT uses XHTML to match nodes for transforming an XML document into a different document.
b) In its most current specification, XSLT does not allow for iteration through a node set returned by an XPath expression.
c) By using XSLT, XML documents can easily be converted between formats.
d) Like element `choose`, element `if` provides a single conditional test.
e) XSLT allows for modularity in stylesheets, which enables XSLT documents to import other XSLT documents.
f) The document resulting from an XSLT transformation may be in the format of an XML document, XHTML/plain text or any other text-based document.
g) XSLT does not provide default templates; all templates must be custom built by the programmer.
h) XSLT provides elements to perform conditional processing, such as `if` elements.
i) XSLT does not provide variables.

D.2 Fill in the blanks in each of the following statements.
a) Attribute ________ defines the XSLT specification used in an XSLT document.
b) An XSLT document is an XML document with a root element ________.
c) XSLT provides the ________ element to allow alternative conditional statements.
d) The letter T in XSLT stands for ________.
e) Templates of an XSLT document can be applied to specific nodes of an element by using element ________.
f) The `template` element matches specific ________ of an XML document.
g) Attribute ________ has values `ascending` and `descending`.
h) Element ________ includes other XSLT documents in the current XSLT document.

**ANSWERS TO SELF-REVIEW EXERCISES**

D.1 a) False. XSLT uses XPath to match nodes when transforming an XML document into a different document. b) False. XSLT allows for iteration through a node set returned by an XPath expression. c) True. d) False. Unlike element `choose`, element `if` provides a single conditional test. e) True. f) True. g) False. XSLT provides several default templates for the nodes of an XML document. h) True. i) False. XSLT provides variables.

D.2 a) `version`. b) `stylesheet`. c) `choose`. d) Transformation. e) `apply-templates`. f) nodes. g) `order`. h) `include`. 
E.1 Introduction

In this appendix, we provide instructions for downloading, installing and configuring Sun Microsystems’s Java 2 Enterprise Edition SDK (J2EE SDK), which is a reference implementation of the J2EE 1.2.1 specification. The J2EE SDK is not intended for use in production environments, because it lacks the advanced features and stability of commercial application servers. Instead, it serves as a tool that developers can use to test applications, and as a guide for application-server vendors when building their own J2EE implementations.

E.2 Installation

The J2EE SDK version 1.2.1 is available for download from java.sun.com/j2ee/j2sdkee. Download the appropriate version for your platform and follow the installation instructions at java.sun.com/j2ee/j2sdkee/install.html. The J2EE SDK requires two environment variables—\texttt{J2EE\_HOME} and \texttt{JAVA\_HOME}. The variables should be set to your J2EE SDK installation directory (e.g., \texttt{c:\j2sdkee1.2.1}) and J2SE SDK installation directory (e.g., \texttt{c:\jdk1.3.1}), respectively. In Windows 2000, specify these environment variables in the System control panel on the Advanced tab. For most UNIX systems, use the following commands to set the environment variables

\begin{verbatim}
J2EE_HOME=/usr/local/j2sdkee.1.2.1; export J2EE_HOME
JAVA_HOME=/usr/local/jdk1.3.1; export JAVA_HOME
\end{verbatim}

Be sure to replace \texttt{/usr/local/} with the name of the directory in which you installed each SDK. If you require database drivers or other packages that are not part of the J2EE
or J2SE SDKs, you also must set the **J2EE_CLASSPATH** environment variable to include those packages.

Throughout this book we use the Cloudscape 3.6 database. The J2EE SDK includes an older version of Cloudscape. To upgrade Cloudscape to the current version, you must replace the JAR files in the `j2sdkee1.2.1\lib\cloudscape` subdirectory with current versions. Copy `client.jar`, `tools.jar` and `cloudscape.jar` from the `Cloudscape_3.6\lib` directory to `j2sdkee1.2.1\lib\cloudscape`. Also, copy `RmiJdbc.jar` from `Cloudscape_3.6\frameworks\RmiJdbc\classes` to `j2sdkee1.2.1\lib\cloudscape`.

### E.3 Configuration

After installing the J2EE SDK, you must configure several options in the SDK’s configuration files. The following subsections provide instructions for configuring data sources and the HTTP server in the J2EE SDK.

#### E.3.1 JDBC Drivers and Data Sources

The configuration file `j2sdkee1.2.1\config\default.properties` contains configuration information for data sources, including URLs, JNDI names, properties and JDBC driver classes. Property **jdbc.drivers** is a colon separated list of JDBC driver class names. For example:

```
jdbc.drivers=COM.cloudscape.core.JDBCDriver:
oracle.jdbc.driver.OracleDriver
```

Note that there should be no spaces in the value of the **jdbc.drivers** property, and all values must appear on a single line. Property **jdbc.datasources** assigns JNDI names, URLs and properties for data sources. The pipe character separates JNDI names and URLs, it also separates each data source entry. For example, the following line includes the JNDI name and URL for the Deitel Bookstore database (Chapters 17–20) and for the **authors** database of Chapter 8. Note that all entries must appear on the same line, and without spaces between entries.

```
jdbc.datasources=jdbc/books|jdbc:cloudscape:rmi:books;create=true|jdbc/Bookstore|jdbc:cloudscape:rmi:Bookstore;create=true
```

#### E.3.2 HTTP properties

HTTP and Web properties are configured in file `web.properties`. Property **http.port** specifies the port number for the HTTP server. The default port is 8000. If you have other services that use port 8000 running on your computer, you may need to change the port value. Property **documentroot** configures the base location from the Web server serves documents. The default directory is `j2eesdk1.2.1\public_html`.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 8/23/01
F.1 Introduction

The Java Community Process (JCP)\(^1\), which started in 1998, is a community-built organization designed to improve the progress and efficiency of Java technologies. Through a multistage process of testing and revision, the JCP creates and updates Java specifications. Along with each specification, the JCP includes a reference implementation and a compatibility test suite. The JCP enables everyone to participate in the development and ratification of the Java Platform. After reading this appendix, you may wish to visit the JCP Web site at [www.jcp.org](http://www.jcp.org) for further details on the JCP and to learn about the various specifications currently under development.

F.2 Participants

The JCP consists of five participants who contribute to the improvement and evolution of each Java specification—the Program Management Office (PMO), the Executive Committees (ECs), Experts, members and the public.

F.2.1 Program Management Office

The Program Management Office (PMO) is the branch of Sun Microsystems that governs and directs JCP. The PMO does not work directly on the specifications, but instead it oversees the work and decisions of the Executive Committee.

F.2.2 Executive Committee

JCP members are elected or ratified to serve three years on an Executive Committee (EC). The primary function of the ECs is to vote on each Java Specification Request (JSR) and the subsequent drafts of the specification. There are two ECs, each covering different Java

---

1. The [jcp.org](http://jcp.org) Web site was our primary resource for information presented in this appendix.
Platforms. One EC is responsible for desktop/server specifications, corresponding to the Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE); this EC currently contains representatives from fifteen organizations and companies including IBM, Apple, and BEA Systems, and one individual. The other EC is responsible for consumer/embedded specifications, corresponding to the Java 2 Micro Edition (J2ME); this EC also consists of sixteen represented companies including Sony, Palm, Motorola, and IBM.

F.2.3 Experts

Every *Java Specification Request (JSR)* has an *Expert Group* associated with it. Experts are nominated representatives from JCP member organizations, or individuals who have completed an Individual Expert Participation Agreement (IEPA). Experts have an advanced knowledge in the material covered by the particular JSR. Individuals may nominate themselves to join an Expert Group or other members may suggest them. Through a series of public and community-restricted drafts, the Expert Group creates the specification proposed in the JSR. Each JSR also has a *Specification Lead (Spec Lead)* associated with it; an Expert from the Expert Group may accept the position and head the formation of the specification. The submitter of the JSR proposal typically becomes the Spec Lead.

F.2.4 Members

JCP Members are companies or individuals who sign a *Java Specification Participation Agreement (JSPA)* and pay a $5000 yearly fee for commercial businesses or a $2000 yearly fee for government, education or non-profit organizations. Membership allows individuals to submit JSRs, review all drafts of the specifications, vote for members of the Executive Committees, and attend Java events. Currently, there are over 400 businesses, including IBM, Bank of America, Motorola, and Xerox, holding a JCP membership.

F.2.5 Public Participation

The public (anyone with Internet access) also has the chance to be part of the development process. Current JSRs and public drafts up for review are available for the public to download, read and comment on via the Internet.

F.3 Java Community Process

The JCP follows a detailed procedure to ensure the practicality and assurance of each specification. Each proposed specification must pass a series of critiques, revisions, and ballots before meeting the corresponding EC’s approval.

F.3.1 Initiation Phase

The introduction of a new specification or the revision of an old one begins with the JCP members. A single member or group of members may produce and submit a JSR to the PMO. Included in the JSR are the names of the members making the request, a proposed Spec Lead and a list of preferred members for the Expert Group. The JSR states the grounds for the proposal, goals for the outcome of the specification and helpful guides to beginning the project.
Upon receiving a JSR, the PMO assigns the JSR to the appropriate EC. By creating a Web page for the JSR, the PMO makes the JSR public. This begins the fourteen day JSR Review process; current JSRs are available at jcp.org/jsr/all. After reading the proposal, the public can submit comments and ideas. The EC then reviews these remarks and posts them on the JSR’s Web Page. During the JSR Review, members have the opportunity to submit applications to the PMO, requesting to join the Expert Group (should the EC approve the JSR).

While the public, Experts and members review the JSR, the EC reads the JSR, considers received comments and votes on the JSR Approval Ballot. After the fourteen day period of the JSR Review, all votes are submitted, and the JSR passes or fails.

If the JSR fails, a new revision period of fourteen days begins. During this time, the PMO sends all of the EC’s comments to the creators of the JSR. The creators may then review the suggestions and criticisms and revise the JSR. When the PMO receives a redrafted edition of the JSR, the PMO posts the fresh copy and the review period begins again. If the revised document fails to receive approval in the JSR Reconsideration Ballot, the failure status is permanent. If the JSR meets approval, the JSR is ready to be developed.

### F.3.2 Community Draft Phase

After the JSR passes either the Approval Ballot or the Reconsideration Ballot, the Expert Group forms. The Spec Lead position is usually filled by the Expert suggested in the JSR. The Expert Group contains members requested in the JSR as well as members who applied after viewing the JSR. Note that there is not a specified size for the Expert Group. The Spec Lead, in accordance with the existing Expert Group, may accept new members, allowing the Expert Group to diversify and gain a broad range of opinions.

The Spec Lead and the Expert Group work with software developers, computer industry workers, educators and other members to form a draft of the specification suitable for the community to review. During the Community Review (a thirty to ninety day period), members and Experts review and comment on the Community Draft.

Throughout the Community Review, the Spec Lead is required to report all changes on the draft to the PMO. The PMO then announces the alterations to the community and posts the latest version of the draft on the JCP Web page.

In addition to the actual specification, JCP provides a Reference Implementation (RI) and a Technology Compatibility Kit (TCK). The RI is an implementation of the specification, confirming that the specification can in fact be implemented. The RI also will act as a guide for others who implement the specification. The TCK is a set of trials, tools and records that implementors of the specification use as a test suite. The Spec Lead is responsible for both producing and licensing the RI and TCK. When the Community Draft released, the Spec Lead provides information on the production and licensing of the RI and TCK to the PMO.

In the final seven days of the Community Review, the EC votes on the Draft Specification Approval Ballot. If the draft fails the vote, the Expert Group has thirty days to read the comments provided by the EC and alter the draft. After a new edition is submitted, the EC votes in the Draft Specification Reconsideration Ballot. If the draft meets the EC’s approval, the Expert Group addresses appropriate feedback from the community review, then makes the draft available for public review.
F.3.3 Public Draft Phase

During the Public Review (a thirty to ninety day period), the draft of the approved specification is posted for both members and the public to view and comment on. The Expert Group considers the received commentary and makes appropriate changes to the draft, posting the latest version of the specification on the corresponding JSR Web page.

F.3.4 Final Phase

The Expert Group has the opportunity to update the draft after the public reviews it. Both JCP members and the public may view the Proposed Final Draft on the JCP Web site.

The Spec Lead must test that it is possible to implement the specification. By providing the RI, the Spec Lead ensures that an implementation of the specification is possible. The Expert Group must confirm that the RI implements the specification adequately, that the TCK provides sufficient test coverage and that the RI passes the tests in the TCK. Any changes to the specification are posted on the JCP Web site and are open to comments. In addition, the PMO requires the Spec Lead to record challenges faced with the tests in the TCK; this information is also posted on the JCP Web site.

The PMO sends the Final Draft to the Executive Committee and announces where to find the RI and TCK. In the Final Approval Ballot, the EC votes on the Final Draft. If the draft fails, the Spec Lead and Expert Group have thirty days to modify the specification, RI and TCK. If a modified specification is submitted, the EC votes on a Final Reconsideration Ballot. If the Final Draft is approved, the Final Draft is posted on the JCP Web site, along with all alterations. The JSR is complete, and the Expert Group is dismissed.

F.3.5 Maintenance Phase

After the specification, RI and TCK are finished and implemented, a Maintenance Period begins. The Maintenance Lead (ML), usually the same person as the Spec Lead, answers questions, clarifies discrepancies and decides if the specification should be updated. The ML considers requests for alterations to the specification and enters them into a Change Log as “proposed” changes. For the following thirty days, anyone can comment on the suggested changes by visiting the JCP Web site.

After the thirty day Maintenance Period the ML categorizes a suggested change as “accepted,” meaning the minor revision will be made, “deferred,” meaning the revision will possibly become a JSR, or “proposed,” meaning the revision will remain as a suggested idea. If the ML terms the revision as “deferred,” the ML may decide to submit the proposed change as a JSR.

During the Maintenance Period a member of the EC may request to change the status of a minor change to a JSR. For this request the PMO will hold an Item Exception Ballot in the last seven days of the Maintenance Period. If the ballot passes the revision will change status to a JSR, otherwise the change remains in the “proposed” category.

Along with the upkeep of the specification, the ML also accordingly updates the TCK and RI. The ML must record challenges of the tests contained in the TCK. If the original TCK implementor is dissatisfied with the change, the implementor may appeal it by sending a written document to the PMO. The EC votes on the complaint in an Appeal Ballot.
Java Native Interface (JNI)

Objectives

- To become familiar with basic JNI concepts.
- To create simple JNI applications.
- To convert C++ and JNI types.
- To learn how to interact with Java objects natively.
- To learn native exception handling.
G.1 Introduction

The Java Native Interface (JNI) allows programmers to access existing libraries written in other languages such as C and C++. JNI allows programmers to use Java without requiring that existing libraries be entirely rebuilt. JNI makes it possible for large companies to migrate to Java without sacrificing the resources required to port existing libraries to Java. JNI also provides the developer with access to system libraries—at the cost of portability. JNI can be useful in time-critical applications—programmers may write a piece of the application in assembly code and link this program with Java to provide better performance. In this appendix, we explain how to integrate Java with C++ libraries. We show how to call native functions stored in compiled libraries from Java, and how to access Java objects, methods and member variables from C++. This appendix assumes a familiarity with C or C++. Although the examples are written in C++, they can be converted to C with little effort. Throughout this appendix we refer to native functions as the routines written in C++ and native methods as the native declarations contained in Java. Native code can be defined as code which can be compiled down to a processor specific binary format.

Performance Tip G.1

Communication between Java and native code is slow, too many JNI calls will actually degrade performance.

G.2 Getting Started with Java Native Interface

Enabling a Java program to call a native function requires several steps. A Java wrapper class is a class that either aggregates methods from disparate sources into a single class or encapsulates a complex API into simple method calls. In the wrapper classes presented in this appendix, we will define native methods that correspond to functions implemented in a C++ library. The javah utility generates header files that enable the Java program to invoke native methods. This utility is part of the J2SE SDK 1.3.1.

Class JNIPrintWrapper (Fig. G.1) is a wrapper class for a native method that prints a simple message to the standard output. It contains an example of the code required to load a library—the compiled code that will be accesses by the Java program (line 8). System.loadLibrary will search your library paths for a library with the given name. Generally, the library should be loaded in a static block. This ensures that the library is loaded before any native methods are called. If the library is not loaded before a native call, the program will exit with a UnsatisfiedLinkError. Line 12 declares method...
printMessage with the keyword `native`, which indicates that the method is implemented in a native library, not in Java. Similar to an `abstract` method, `native` method definition is never provided in Java.

JNI needs specific header files to interact with native code. The JDK includes the tool `javah` to produce ANSI C header files for use with native C and C++ code. The command line indicates that `javah` should create `JNI header files`. You must run `javah` on each compiled Java class that contains a `native` method. Do not edit these headers—if you make a change to the wrapper class, run `javah` again on the compiled class to generate appropriate header files. The following command:

```
javah -jni JNIPrintWrapper
```

creates the `JNIPrintWrapper.h` header file. If `JNIPrintWrapper` were in a package named `utilities.native` the resulting file would be named `utilities_native_JNIPrintWrapper.h`. Note that the `-jni` option specifies that headers are produced for JNI.¹

Header `JNIPrintWrapper` (Fig. G.2) is the header that `javah` creates for Java class `JNIPrintWrapper`. Line 2 includes `jni.h`, the header that contains `JNI helper functions` required for the C++ library to interact with the Java. JNI requires the use of C-style function identifiers, thus, the preprocessor commands on lines 7–9 ensure the native functions are compiled correctly. The comments on lines 10–14 represent information needed by JNI to use the methods. The signature (line 13) indicates the argument and the return types of the function. Lines 15–16 contain two platform specific `#defines`—`JNEXPORT` and `JNIACALL`, each help handle JNI interactions with the C++ code for your system. The function name consists of `Java_calling class_methodname`. Function `printMessage` takes three arguments. The first argument, of type `JNIEnv` is a pointer to a table of functions that assist the native code’s interaction with Java. `JNIEnv` is always passed as the first argument to a JNI function. In this appendix, `JNIEnv` variables are accessed using C++. In C++ a `JNIEnv*` pointer would be accessed like an object:

```
env->functionName(args)
```

```
1   // Fig. G.1 JNIPrintWrapper.java
2   // Allows access to native methods
3
4   public class JNIPrintWrapper {
5
6       // load library JNIPrintLibrary into JVM
7       static {
8           System.loadLibrary( "JNIPrintLibrary" );
9       }
10
11      // native C++ method
12      public native void printMessage( String message );
13   }
```

---

1. JNI was introduced in JDK 1.1—before that developers used a more cumbersome native interface, the headers for this older style may be generated with the `-stubs` option.
In C, it would be accessed using the syntax:

\[
(*env)\rightarrow\text{functionName(env, args)}
\]

The second argument sent to `printMessage` is a reference to the Java object that called this native function. Any arguments that follow are the values passed to the function from the Java code.

Before using this header you must make sure that your compiler or IDE knows where to look for `jni.h` and the `#define`s. They are located in the following directories:

```
JDK directory\include\your platform
```

Java primitive types and object types do not map directly to those used in C/C++. To pass these types between languages, JNI includes a predefined set of types. All Java objects are passed as `jobjects`, all `Strings` are passed as `jstrings`, and primitives are passed as `jtype` (e.g., `jint` and `jdouble`). Conversion between C/C++ and JNI primitive types is handled transparently. However, working with `jobjects`, including arrays is more complicated.

The implementation of the `javah` header (Fig. G.3) `JNIPrintWrapper-Impl.cpp` converts a `jstring` into a `char *` and uses C++ to output the string. Java uses `Unicode` (see Appendix I) to represent its characters, however C++ uses an `ANSI` format (Latin-1). Consequently, a direct assignment is not possible. Unicode may be converted into an ANSI comparable format called `UTF-8`, which C++ supports. JNI performs conversion with the function `GetStringUTFChars`, which returns a pointer to the string in UTF-8 format. JNI sets `jboolean` variable `copied` to `JNI_TRUE` if the string passed
Appendix G

Java Native Interface (JNI) 1709

Appendix G

is a copy of the Java string and JNI_FALSE if it is a pointer to the original string. We dis-
cuss the differences between a values vs. a reference to a string or array in Section G.4. In
either case the returned string should be assigned to a char*. After displaying the string
release the memory used by the UTF-8 string. If the string is not released it will result in a
memory leak. Releasing the string requires a call to ReleaseStringUTFChars, with
the jstring and char* as arguments.

After creating the library and the wrapper class the native code must be loaded into the the Java JVM. On Win32 platforms, libraries have the extension .dll. To avoid a Java UnsatisfiedLinkError the path to the library must be set with this command:

set PATH=%PATH%;library path

On Solaris, with csh, the library must be named liblibraryname.so and the library path is set with:

setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:library path

Class JNIPrintMain (Fig. G.4) demonstrates the main class of the JNI application. The body of the main method (lines 14–17) demonstrates the benefit of the wrapper class. Within the main Java application, native calls written in the same manner as other method calls.
G.3 Accessing Java Methods and Objects from Native Code

JNI can do far more with native code than just sending and receiving arguments. You may access Java methods directly from the native library, create new Java objects and even create a new instance of the JVM. This section includes demonstrations of method calling, object creation and array handling, directly from native code.

Class JNIMethodWrapper (Fig. G.5) behaves similarly to the previous wrapper class, except that it contains two methods. In the first native function we will call a method on the PIContainer class passed to the function (line 12). In the second method, we do not pass an instance of PIContainer, it is created in native code (line 13).

```
// Fig. G.4 JNIPrintMain.java
// Creates a new instance of the Java wrapper class,
// and calls the native methods.

public class JNIPrintMain{

    // uses JNI to print a message
    public static void main( String args[] )
    {
        JNIPrintWrapper wrapper = new JNIPrintWrapper();

        // call to native methods through JNIWrapper
        wrapper.printMessage( "Welcome to JNI!\n" );
    }
}
```

Welcome to JNI!

```
// Fig. G.5 JNIPIWrapper.java
// Allows for access to native function.

public class JNIPIWrapper {

    // loadJNIPLibrary into JVM
    static {
        System.loadLibrary( "JNIPILibrary" );
    }

    // native c++ methods
    public native double getPI( PIContainer container );
    public native double getPI();
}
```

Fig. G.4 JNIPrintMain calls the native method via the wrapper class.

Fig. G.5 JNIPIWrapper encapsulates the native methods and loads the library.
Class **PIContainer** (Fig. G.6) contains method **getPI** that returns the value of the constant **PI**. The native code uses **PIContainer** to demonstrate the construction of Java objects and the invocation of methods.

**JNIPIWrapper.h** (Fig. G.7) is the **javah**-generated header for our library, which includes the return type **jdouble** (line 15).

```java
public class PIContainer {
    public double getPI() {
        return Math.PI;
    }
}
```

**Fig. G.6** **PIContainer** returns the **PI** member of **java.lang.Math**.

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JNIPIWrapper */

#ifndef _Included_JNIPIWrapper
#define _Included_JNIPIWrapper
#ifdef __cplusplus
extern "C" {
#define __cplusplus
JNIEXPORT jdouble JNICALL Java_JNIPIWrapper_getPI__ (JNIEnv *, jobject);

JNIEXPORT jdouble JNICALL Java_JNIPIWrapper_getPI__LPIContainer_2 (JNIEnv *, jobject, jobject);

#undef __cplusplus
}
#endif
#endif
```

**Fig. G.7** **JNIPIWrapper.h** is the **javah** generated header file for the native functions.

C++ file **JNIPIWrapperImpl.cpp** (Fig. G.8) uses JNI to create and invoke native methods and objects. Lines 16–22 demonstrate invoking a method on an object that has already been created. **GetObjectClass** in line 16 takes an argument of type **jobject** and returns the **jclass** to which it belongs. The argument in this case is the **jobject** passed by the calling method.

```cpp
// Fig. G.8 JNIPIWrapperImpl.cpp
// Calls methods on the PIContainer object sent to it.
// Creates new Java objects.

// standard c header
#include <iostream.h>

// header produced by javah
#include "JNIPIWrapper.h"

// retrieves value of PI based on passed PIContainer
JNIEXPORT jdouble JNICALL Java_JNIPIWrapper_getPI__LPIContainer_2
(JNIEnv * env, jobject thisObject, jobject containerObject)
{
    // retrieve PIContainer class
    jclass PIClass = env->GetObjectClass( containerObject );

    // retrieve method ID of getPI
    jmethodID mid = env->GetMethodID( PIClass, "getPI", "()D" );

    // call method getPI of class PIContainer with no arguments
    jdouble PI = env->CallDoubleMethod( containerObject, mid, NULL );

    return PI;
}

// retrieves value of PI by creating new PIContainer
JNIEXPORT jdouble JNICALL Java_JNIPIWrapper_getPI__
(JNIEnv * env, jobject thisObject ) {
    // finds Class PIContainer
    jclass PIClass = env->FindClass( "PIContainer" );

    // retrieves ID of constructor
    jmethodID constructorID =
        env->GetMethodID( PIClass, "<init>", "()V" );

    // creates new PIContainer
    jobject newContainer =
        env->NewObject( PIClass, constructorID, NULL );

    // retrieves method ID of getPI
    jmethodID mid = env->GetMethodID( PIClass, "getPI", "()D" );
```
Function `GetMethodID` (line 19) requires a `jclass` object to determine the method's `jmethodID`. A `method ID` is a unique ID created and associated with each class method. Function `GetMethodID` also requires the name of a method within the `jclass` object and a signature string specifying the arguments and return type of that method. If the specified method cannot be found, the method ID returned is zero. The signature must be in the format:

$$( \langle \text{argument types}\rangle ) \langle \text{return type}\rangle$$

Primitive types are usually specified by their uppercase first letter (e.g., $D$ for `double`, $I$ for `int` and $V$ for `void`). Exceptions to this rule are $J$ for `long` and $Z$ for `boolean`. Object types must be prefixed by $L$ followed by the class, including packages and terminated by a semicolon. For example, $Ljava/lang/String;$ specifies the `String` type. Including an open bracket ($[$) before a signature indicates an array, $(D)V$ represents a method that requires an array of `doubles` and returns `void`. The full listing of type mappings is shown in Fig. G.9.

Line 22 calls the function `CallDoubleMethod` to invoke the Java method `getPI`. Methods can be called with `Call` type `Method`, where `type` represents the return type. `CallObjectMethod` can be called for any Java object. The first argument is a reference to the object containing the method, the second contains the method ID and any additional arguments are parameters to send to the method.

<table>
<thead>
<tr>
<th>Type</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>B</td>
</tr>
<tr>
<td>char</td>
<td>C</td>
</tr>
<tr>
<td>double</td>
<td>D</td>
</tr>
<tr>
<td>float</td>
<td>F</td>
</tr>
<tr>
<td>int</td>
<td>I</td>
</tr>
<tr>
<td>long</td>
<td>J</td>
</tr>
<tr>
<td>short</td>
<td>S</td>
</tr>
<tr>
<td>void</td>
<td>V</td>
</tr>
<tr>
<td>boolean</td>
<td>Z</td>
</tr>
<tr>
<td>class</td>
<td>$L/full/class/name;$</td>
</tr>
<tr>
<td>array of type</td>
<td>$[type$</td>
</tr>
</tbody>
</table>

Fig. G.8 | JNIWrapperImpl.cpp demonstrates method calls and object construction (part 2 of 2).

Fig. G.9 | Signature type mappings.
The second function, lines 28–49 creates a new `PIContainer`, then retrieves its value of `PI`. Line 32 demonstrates the use of JNI function `FindClass`, which returns a reference to any fully qualified class name. If you do not have an object instance, call function `FindClass` from native code to retrieve the `jclass` object. After obtaining the class, a new instance must be created. The first step is to obtain the method ID of `<init>`, the method name used in JNI to signify a constructor. The method ID, along with the class, and any constructor arguments must be used in the `NewObject` function to create a new object of that class (line 40).

Class `JNIPIMain` (Fig. G.10) uses the wrapper class to retrieve `PI`. It creates a new instance of `PIContainer` (line 12) for native use. The class displays the retrieved `PI` value to demonstrate correct return value. Method `getPI` is then called again without passing any arguments.

Next, we will demonstrate how to call static methods. An instance of a static member cannot be obtained—thus the `jobject` which refers to an instance of an object cannot be used. The static JNI functions require the `jclass` of the object, which simply represents the class and contains any static members.

Class `JNIStaticWrapper` (Fig. G.11) is the wrapper class for gaining access to static methods and variables. The native method in this class will receive static variables from class `MathConstants`.

Class `MathConstants` (Fig. G.12) is a container class for the member variables of class `Math. MathConstants` methods and variables will be accessed from native code. `JNIStaticWrapper.h` (Fig. G.13) is the `javah` generated header file. The signature on line 12 specifies that the method must take an argument of type `MathConstants`.

```java
// Fig. G.10 JNIPIMain.java
// Loads the native library, creates a new instance of the
// Java wrapper class and uses native code to call getPI.

public class JNIPIMain {

    public static void main( String args[] )
    {
        JNIPIWrapper wrapper = new JNIPIWrapper();

        // native code retrieves PI from instance of PIContainer
        double pi = wrapper.getPI( new PIContainer() );
        System.out.println( "PI is " + pi );

        // native code retrieves PI, creates its own PIContainer
        double pi2 = wrapper.getPI();
        System.out.println( "PI2 is " + pi2 );
    }
}
```

```
PI is 3.141592653589793
PI2 is 3.141592653589793
```

Fig. G.10  `JNIPIMain` calls each native method via the wrapper class.
Appendix G

Java Native Interface (JNI) 1715

Fig. G.11 JNIStaticWrapper.java

public class JNIStaticWrapper {
    // load library JNIMathLibrary into JVM
    static {
        System.loadLibrary( "JNIMathLibrary" );
    }
    // native C++ method
    public native void printStaticMembers( MathConstants constants );
}

Fig. G.11 JNIStaticWrapper loads JNIMathLibrary and declares native method printStaticMembers.

Fig. G.12 MathConstants.java

public class MathConstants {

    public static final double PI = Math.PI;
    public final double E = Math.E;

    public static double getPI()
    {
        return PI;
    }
}

Fig. G.12 MathConstants contains common math constants from Math.

Fig. G.13 JNIStaticWrapper.h javah generated header file (part 1 of 2).
JNIStaticWrapperImpl.cpp (Fig. G.14) demonstrates native calls to static and non-static members from native code. Accessing member variables follows the JNI format—obtain the jclass, get the member ID and access the member. Lines 19–25 show these steps to access a static variable. GetFieldID obtains the jfieldID of the variable you wish to access. A jfieldID is the unique ID of a variable field within an object. The jfieldID can be used to locate a member variable the same way a jmethodID can locate a method. You must specify the class, the variable name and the field signature. Field signatures correspond to method signatures—however, only a single type must be specified. GetTypeField returns the non-static member contained within the given object instance that matches the given jfieldID. To obtain the jfieldID of a static member requires a call to GetStaticFieldID and requires the same arguments as a call to a non-static field (line 30). Retrieving the field requires a call to GetStaticTypeField, and passes jclass as the first argument (line 31–32). To set member fields call SetTypeField, with a jobject instance, the fieldID and a value of type as arguments. Setting static fields requires a call to SetStaticTypeField with the jclass as the first argument. Invoking static methods uses the same technique: specify GetStaticMethodID, and call CallStaticTypeMethod with the jclass as an argument.

Fig. G.14 JNIStaticWrapperImpl.cpp accesses and prints static members of the given MathConstants class (part 1 of 2).
Class **JNIStaticMain** (Fig. G.15) is the main class for this example, it loads the library, creates the wrapper and calls the **native** method.

```java
// Fig. G.15 JNIStaticMain.java
// creates a new instance of the Java wrapper class and calls to
// print the static members of the MathConstants class.

public class JNIStaticMain {

    public static void main( String args[] )
    {
        JNIStaticWrapper wrapper = new JNIStaticWrapper();

        // access static members from MathConstants
        wrapper.printStaticMembers( new MathConstants() );
    }
}
```

Value of E in MathConstants is 2.71828
Value of PI in MathConstants is 3.14159

**Fig. G.15** **JNIStaticMain** prints **static** math constants via the wrapper class.
G.4 JNI and Arrays

Converting arrays from Java to native code is complex because, as opposed to native languages, elements in Java arrays are not always placed together in contiguous memory. Thus, when a native function requires access to an array created in Java, JNI must organize the array correctly, then it must ensure the array is returned back to Java when the native function completes.

JNIArrayWrapper (Fig. G.16) contains functionality to print the array returned by a native function. Method newRandomArray creates an array of random integers of length size.

JNIArrayWrapper.h (Fig. G.17) declares the function with a return type of jintArray, which is an array of integers. The suffix Array is added to any JNI type to specify a Java array.

```
1 // Fig. G.16 JNIArrayWrapper.java
2 // Allows access to native methods.
3
4 public class JNIArrayWrapper {
5     // load library JNIArrayLibrary into JVM
6     static {
7         System.loadLibrary( "JNIArrayLibrary" );
8     }
9
10     // native C++ method
11     public native int[] newRandomArray( int size );
12
13     // Java method that calls native method
14     public void outputRandomNumbers( int amount )
15     {
16         int randomNumbers[] = newRandomArray( amount );
17
18         for ( int i = 0 ; i < amount ; i++ )
19             System.out.println( randomNumbers[ i ] +
20                             " random number " + i );
21     }
22 }
```

Fig. G.16 JNIArrayWrapper loads JNIArrayLibrary and displays the numbers in the returned array.

```
1 /* DO NOT EDIT THIS FILE - it is machine generated */
2 #include <jni.h>
3 /* Header for class JNIArrayWrapper */
4
5 #ifndef _Included_JNIArrayWrapper
6 #define _Included_JNIArrayWrapper
7 #ifdef __cplusplus
8     extern "C" {
9 #endif
```

Fig. G.17 JNIArrayWrapper.h javah generated header file (part 1 of 2).
Fig. G.18  JNIArrayWrapperImpl.cpp demonstrates primitive and Object array creation and control (part 1 of 3).

C++ file JNIArrayWrapperImpl.cpp (Fig. G.18) creates two arrays, one of primitive type `jintArray`, and the other a `jobject` array containing `Integer` objects. Lines 29–30 create a new instance of class `Integer`. Line 33 invokes JNI function `NewObjectArray` to create the `Integer` array. Function `NewObjectArray` takes as arguments the array size, the array type and the initial value for each array element. Creating a primitive integer array works similarly; however, function `NewIntArray` requires one argument, the size, which is the size of the array. Each of these functions builds a Java array. Accordingly, the elements may or may not be created contiguously in memory. To obtain access to individual elements of a primitive array within native code requires a call to `GetprimitiveArrayElements`, with the `jtypeArray` and a `jboolean` as arguments. The `jboolean isCopy` is passed by reference (line 40), and is set by the function to either `JNI_TRUE` or `JNI_FALSE`. If the array is laid out in contiguous memory the resulting `jboolean` has the value `JNI_FALSE` which indicates that the native code has direct access to the array. However, `JNI_TRUE` indicates that a copy of the array was created. If we have received a copy, we must be careful to release the memory, and be sure the array is updated in Java. Lines 61–63 determine how the array was received, if it is a copy it is released with the call `Release-typeArrayElements`. Releasing an array requires passing the array, our pointer to the array, and the way it is to be released. We set the final argument to 0 in line 62—this tells JNI to release the memory and update the original array in Java. Other options include `JNI_ABORT` which frees the memory but not update Java and `JNI_COMMIT` which will update Java but does not release the memory. Currently there is no documentation on how to release the memory manually, thus `JNI_COMMIT` is not recommended.
// header produced by javah
#include "JNIArrayWrapper.h"

// creates and fills two arrays
JNIEXPORT jintArray JNICALL Java_JNIArrayWrapper_newRandomArray(
    JNIEnv * env, jobject thisObject, jint size )
{
    // boolean to determine if elements are copied
    jboolean isCopy;
    jclass integerClass;
    jobjectArray objectArray;
    jobject randomIntegerObject;
    jmethodID mid;
    int randomInt;

    // seed rand function with time
    srand( time( NULL ) );

    // locate object and constructor
    integerClass = env->FindClass( "java/lang/Integer" );
    mid = env->GetMethodID( integerClass, "<init>", "(I)V" );

    // build new Integer array of length size
    objectArray = env->NewObjectArray( size, integerClass, NULL );

    // create new primitive int array
    jintArray intArray = env->NewIntArray( size );

    // return pointer to array of elements
    jint * intArrayPointer =
        env->GetIntArrayElements( intArray, &isCopy );

    // fill arrays with random numbers
    for ( int i = 0 ; i < size ; i++ ) {
        // create random int
        randomInt = rand() % 100;

        // create Integer containing random int
        randomIntegerObject =
            env->NewObject( integerClass, mid, randomInt );

        // add element to Integer array
        env->SetObjectArrayElement( objectArray, i, randomIntegerObject );

        // assign random int to primitive array
        intArrayPointer[ i ] = randomInt;
    }
}

Fig. G.18  JNIArrayWrapperImpl.cpp demonstrates primitive and Object array creation and control (part 2 of 3).
Lines 39–54 populate both arrays with a set of random numbers. Function `NewObject` creates a new `Integer` object. However, unlike a primitive type, we do not have a handle to the array in a format that C++ can use. We cannot directly convert a Java object into a C++ object, thus we must access it in the Java memory space through JNI. JNI includes the functions `SetObjectArrayElement` (line 49) and `GetObjectArrayElement` to provide access to an object array. The first argument is the array of objects, the third is the object to add to the array, and the second is the index at which to place the object in the array. There is no need to release object arrays, they are never copied from the JVM’s memory space, thus Java’s built-in garbage collector will manage the object memory allocation.

`JNIArrayMain` (Fig. G.19) is similar to the other main classes, it instatiates a wrapper object and calls the `native` method (line 19).

```
public class JNIArrayMain {

   public static void main( String args[] ) {
      JNIArrayWrapper wrapper = new JNIArrayWrapper();

      // outputs ten random numbers
      wrapper.outputRandomNumbers( 10 );
   }
}
```

(continued on next page)
G.5 Handling Exceptions with JNI

Handling exceptions natively can be essential when integrating native code with Java. JNI allows native functions to catch exceptions themselves, or to throw exceptions back to Java. The example in this section uses JNI to do image processing. A resource intensive task such as image processing might gain performance via JNI. However, in our demonstration on exception handling we use several JNI calls, which significantly reduces performance.

The application calls method ImageSizeException (Fig. G.20) if there is an attempt to process an image that is too large.

The application calls exception PixelTintException (Fig. G.21) if the parameters entered for color tinting would increase the color value over an allowable value—255.

```
1 // Fig. G.20 ImageSizeException.java
2 // thrown when the image to be processed is too large.
3
4 public class ImageSizeException extends Exception {
5     // calls parent constructor
6     public ImageSizeException( String message )
7     {
8         super( message );
9     }
10 }
```

Fig. G.20 ImageSizeException used when image is too large.

```
1 // Fig. G.21 PixelTintException.java
2 // extends exception, called when a pixel cannot be fully tinted.
3
4 public class PixelTintException extends Exception {
5     // calls parent constructor
6     public PixelTintException( String message )
7     {
8         super( message );
9     }
10 }
```

Fig. G.21 PixelTintException is used for invalid pixel tint values.
Class **JNITintWrapper** (Fig. G.22) calls the **loadLibrary** method to load the native library **JNITint** (line 8). It also declares the **native** method **tintImage**. This method takes an array of RGB color values and tint the color based on **rTint**, **gTint** and **bTint** (lines 12–14). This method throws an **ImageSizeException**, which ensures that the exception declared in the native code is handled by Java.

**Good Programming Practice G.1**

*Specify the exceptions a native call throws in the wrapper class, this ensures that all calls catch these exceptions.*

Header file **JNITintWrapper.h** (Fig. G.23) contains the JNI function declaration for **tintImage**. The arguments passed are the JNI environment pointer, a reference to the calling object, the RGB color array, the length of the array, the red tint, green tint and blue tint.

```java
// Fig. G.22 JNITintWrapper.java
// loads library JNITint and wraps the native tintImage function.

public class JNITintWrapper {

    // load library JNITint into JVM
    static {
        System.loadLibrary( "JNITint" );
    }

    // tints image based on tint values
    public native int[] tintImage(
        int[] image, int length, int rTint, int gTint, int bTint )
        throws ImageSizeException;
}

Fig. G.22 JNITintWrapper loads the native library and wraps the native function.
```

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JNITintWrapper */

#ifndef _Included_JNITintWrapper
#define _Included_JNITintWrapper
#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jintArray JNICALL Java_JNITintWrapper_tintImage(
    JNIEnv *, jobject, jintArray, jint, jint, jint, jint);

Fig. G.23 JNITintWrapper.h javah generated JNI header file (part 1 of 2).
```
JNITintImage.cpp contains function tintImage, which separates the color values from the int array and increases each by the tint amount to provide a tinting effect (Fig. G.24). This is not the only, or most accurate method of tinting, but it is simple. Line 7 defines MAX_IMAGE_SIZE as 65536, the value of a 256x256 pixel image. Lines 18–28 check the size of the image, and throw an exception if it is too large. Line 21 assigns the jclass of ImageSizeException to sizeException with the FindClass function. Function ThrowNew throws an exception of the provided jclass type, with the given message. After throwing an exception, tintImage exits (line 27), any thrown exception declared in native code will be handled in Java if it is not cleared. Lines 52–106 contain the color tinting code. The ARGB integers passed to this function are in sRGB format. This 32 bit format allocates 8 bits for each element, Alpha (transparency), red, green then blue, in that order. We use bit masking to separate the colors from the combined integer (line 54–61). Then, we shift the bits of the tint values to match the color location. The colors are incremented by the tint values to produce a tinting effect. (lines 63–65).

Common Programming Error G.1
When an exception has been thrown, do not make any JNI function calls, except those that deal with exceptions, until the flag has been cleared. Further calls may lead to unpredictable results, including crashing the JVM. Check for any exception that can be fixed natively, clear the exception, then fix it.
// obtain jclass ImageSizeException
sizeException = env->FindClass( "ImageSizeException" );

// throw exception
env->ThrowNew( sizeException, "Image is too large" );

// return and allow Java to handle the exception
return NULL;

// obtain jclass of PixelTintException for use later
pixelTintException = env->FindClass( "PixelTintException" );

jthrowable exception;
int warningCount = 0;
jboolean isCopy, isCopy1;
unsigned int red, blue, green;

// create new array of size length
jintArray warningArray = env->NewIntArray( length );

// obtain pointer to the array object
jint * warningArrayPointer =
  env->GetIntArrayElements( warningArray, &isCopy );

// storage location for RGB array elements
long* elements;

// points elements to the integer array
elements = env->GetIntArrayElements( imageRGB, &isCopy1 );

for ( int i = 0; i < length; i++ ) {

  // determine red element by masking
  red = elements[ i ] & 0xFF0000;

  // determine green element by masking
  green = elements[ i ] & 0xFF00;

  // determine blue element by masking
  blue = elements[ i ] & 0xFF;

  red += rTint << 16;
  green += gTint << 8;
  blue += bTint;

  // throw exception if red value is too large
  if ( red > 0xFF0000 ) {
    env->ThrowNew( pixelTintException, "red value reduced to 255" );
  }
red = 0xFF0000;

// throw exception if green value is too large
if ( green > 0xFF00 ) {
    env->ThrowNew(pixelTintException, "green value reduced to 255");
    green = 0xFF00;
}

// throw exception if blue value is too large
if ( blue > 0xFF ) {
    env->ThrowNew(pixelTintException, "blue value reduced to 255");
    blue = 0xFF;
}

// if an exception occurs store it in exception
if ( ( exception = env->ExceptionOccurred() ) != NULL ) {

    // if exception was of type pixelTintException record
    // the index in an array and clear the exception
    if ( env->IsInstanceOf(exception, pixelTintException) == JNI_TRUE ) {
        warningArrayPointer[ warningCount ] = i;
        warningCount++;
        env->ExceptionClear();
    }

    // else we do not know the exception type, return.
else
    return warningArray;

    // shift bits to recreate pixel
    elements[ i ] = 0xFF000000 | red | green | blue;

    // if first array is copy, release it
if ( isCopy )
    env->ReleaseIntArrayElements(warningArray, warningArrayPointer, 0);

    // if second array is copy, release it
if ( isCopy1 )
    env->ReleaseIntArrayElements(imageRGB, elements, 0);

    if ( warningCount != 0 )
        return warningArray;

Fig. G.24  JNITintImages.cpp tints an array of sRGB color values to demonstrate exception handling (part 3 of 4).
After tinting the values it is possible for a color value to be larger than 255. In this example, we wish to record all pixels that could not be fully tinted. Thus for each value over 255 we throw an exception, and reset the value to 255 (lines 68–84). These exceptions are used as warnings. We do not want the program to terminate when it returns to Java code, to avoid this we clear the exception. To determine if an exception has been thrown, we call JNI function `ExceptionOccurred`. This function returns a `jthrowable` instance of the exception or `JNI_NULL` if an exception was not thrown. We check for the exception (line 87) and then determine the type of the exception using JNI function `IsInstanceOf`, which returns `JNI_TRUE` if the first argument is an instance of the second argument. After determining the type of exception (lines 91–92), we store the current index and make a call to JNI function `ExceptionClear` which clears any thrown exceptions. If possible, you should clear the exception before making JNI calls. Many JNI function calls will crash the JVM if an exception has been thrown.

If the exception does not match our defined exception types, the native function is exited and control returns to Java, where the exception will be handled (lines 99–100). After setting the colors we rebuild the color value by using the `or` operator (lines 104–105). Then, we determine if any of arrays that we have been manipulating are copies, if so we release them back to the Java memory space (lines 109–115). Finally, we return the warning array if it is not empty (lines 117–124).

Class `JNIPanel` extends `JPanel` to provide a display surface for our images (Fig. G.25). The constructor initializes the panels, buttons and sliders in lines 26–91. It loads the image with the `Toolkit` method `getImage`, then uses a `MediaTracker` object to ensure the image is completely loaded before continuing (lines 93–106). We then create a `BufferedImage` with the same dimensions as the loaded image, and assign it a ARGB color format (lines 110–112). We then obtain the graphics context of the `BufferedImage` and write the image’s contents onto it (lines 115–118). Lines 124–126 obtain the array of pixels in the sRGB format and store them in `imageRGB`. Lines 129–131 create a new `BufferedImage` that contains the tinted color values. Finally, the constructor calls method `updateTint`—to processes the image. Method `updateTint` (lines 148–181) retrieves the values of the slider and calls the `native` tinting method. First, a copy is made of the array of colors, this copy allows us to maintain an unmodified copy of the original color values. It calls the `native` method, then prints a warning if any pixels could not be fully tinted, and catches the `ImageSizeException` (lines 154–171). Method `actionPerformed` simply calls `updateTint` when the button is pressed (lines 183–189). Finally, we override the `paintComponent` method of `JPanel` to display our images (lines 138–145).
// Fig. G.25 JNIPanel.java
// Loads an image, creates a copy and uses JNI to tint the image.

public class JNIPanel extends JPanel implements ActionListener {

    BufferedImage loadedImage;
    BufferedImage tintedImage;
    int[] imageRGB;
    int[] tintedRGB;
    JSlider tintRed, tintGreen, tintBlue;
    JButton tintButton;
    JNITintWrapper imageProcess;

    public JNIPanel( String file, LayoutManager layout ) {
        super( layout );
        setPreferredSize( new Dimension( 640, 425 ) );

        // create image manipulation components
        tintRed = new JSlider( 0, 255, 0 );
        tintBlue = new JSlider( 0, 255, 0 );
        tintGreen = new JSlider( 0, 255, 0 );
        tintButton = new JButton( "Tint Image" );

        JLabel redLabel = new JLabel( "Red" );
        JLabel greenLabel = new JLabel( "Green" );
        JLabel blueLabel = new JLabel( "Blue" );

        // set properties for red slider
        tintRed.setMajorTickSpacing( 50 );
        tintRed.setMinorTickSpacing( 10 );
        tintRed.setPaintTicks( true );
        tintRed.setPaintLabels( true );

        // set properties for green slider
        tintGreen.setMajorTickSpacing( 50 );
        tintGreen.setMinorTickSpacing( 10 );
        tintGreen.setPaintTicks( true );
        tintGreen.setPaintLabels( true );

        // set properties for blue slider
        tintBlue.setMajorTickSpacing( 50 );
        tintBlue.setMinorTickSpacing( 10 );
        tintBlue.setPaintTicks( true );
        tintBlue.setPaintLabels( true );

        // obtains image, processes it and displays warnings
    }

Fig. G.25 JNIPanel creates the application GUI and calls the native method (part 1 of 4).
tintColor.setActionCommand( "tint" );

// use this class to process action
tintColor.addActionListener( this );

// create new panels
JPanel sliderPanel =
    new JPanel( new FlowLayout( FlowLayout.LEFT ) );

JPanel redPanel = new JPanel( new FlowLayout() );
redPanel.setPreferredSize( new Dimension( 200, 250 ) );

JPanel greenPanel = new JPanel( new FlowLayout() );
greenPanel.setPreferredSize( new Dimension( 200, 250 ) );

JPanel bluePanel = new JPanel( new FlowLayout() );
bluePanel.setPreferredSize( new Dimension( 200, 250 ) );

// add the components
redPanel.add( tintRed );
redPanel.add( redLabel );
greenPanel.add( tintGreen );
greenPanel.add( greenLabel );
bluePanel.add( tintBlue );
bluePanel.add( blueLabel );
sliderPanel.add( redPanel );
sliderPanel.add( greenPanel );
sliderPanel.add( bluePanel );

// set panel size
sliderPanel.setPreferredSize( new Dimension( 650, 75 ) );

add( sliderPanel );

// add components to JNIPanel
add( tintButton );

MediaTracker tracker = new MediaTracker( this );

// load image
Image image = Toolkit.getDefaultToolkit().getImage( file );

// add image to tracker
tracker.addImage( image, 0 );

try{
    // wait until image is loaded
    tracker.waitForID( 0 );
}

Fig. G.25  JNIPanel creates the application GUI and calls the native method (part 2 of 4).
try {
  super.paintComponent( g );
  Graphics2D graphics = ( Graphics2D )g;
  graphics.drawImage( loadedImage, null, 150, 150 );
  graphics.drawImage( tintedImage, null, loadedImage.getWidth() + 160, 150 );
}

// tints image and displays warnings
public void updateTint()
{
  // get copy of original RGB values
tintedRGB = ( int[] )imageRGB.clone();
  try {
    // try to tint image
    tintedRGB = tintedRGB( 0, 0, tintedRGB, 0, tintedRGB.length);
    // update BufferedImages
    Graphics2D graphics = ( Graphics2D )loadedImage.getGraphics();
    graphics.drawImage( loadedImage, 0, 0, this );
    graphics.drawImage( tintedImage, 150, 150, null, null, tintedImage.getWidth(), tintedImage.getHeight());
  } catch ( InterruptedException e ) {
    e.printStackTrace();
  }
  // create new buffered image
  loadedImage = new BufferedImage( image.getWidth( this ), image.getHeight( this ), BufferedImage.TYPE_INT_ARGB );
  // get graphics context of BufferedImage
  Graphics2D graphics = ( Graphics2D )loadedImage.getGraphics();
  // draw Image on BufferedImage
  graphics.drawImage( image , 0, 0, this );
  // create new native wrapper
  imageProcess = new JNITintWrapper();
  // obtains array of RGB integer values
  imageRGB = loadedImage.getRGB( 0 , 0 , loadedImage.getWidth(), loadedImage.getHeight(), imageRGB, 0, loadedImage.getWidth() );
  // create new buffered image for new RGB values
  tintedImage = new BufferedImage( loadedImage.getWidth(), loadedImage.getHeight(), BufferedImage.TYPE_INT_ARGB );
  // process image and print any warnings
  updateTint();
}

Fig. G.25 JNIPanel creates the application GUI and calls the native method (part 3 of 4).
Class **JNIImageFrame** extends **JFrame** to allow us to add **JNIPanel** object and serves as an entry point for the application (Fig. G.26). The constructor creates a new **JNIPanel** and adds it to the frame inside a **JScrollPane**.

---

Fig. G.25 **JNIPanel** creates the application GUI and calls the **native** method (part 4 of 4).

```java
// call to native code for image processing tint 20 green
int[] warnings =
    imageProcess.tintImage(
        tintedRGB, tintedRGB.length, tintRed.getValue(),
        tintGreen.getValue(), tintBlue.getValue());

if ( warnings != null )
    System.out.println(
        "Not all pixels could be fully tinted" );

// if image is too large catch exception and end program
} catch ( ImageSizeException sizeException ) {
    System.out.println(
        "Image is too large, error was:" );
    sizeException.printStackTrace();
}

// set RGB values
tintedImage.setRGB( 0, 0, tintedImage.getWidth(),
    tintedImage.getHeight(), tintedRGB, 0,
    tintedImage.getWidth() );

// repaint image
repaint();

// handles button events
public void actionPerformed( ActionEvent  e ) {
    // if button pressed then update image
    if ( e.getActionCommand().equals( "tint" ) )
        updateTint();
}

Fig. G.26 **JNIImageFrame** serves as an entry point for the application (part 1 of 2).

```
public class JNIImageFrame extends JFrame {

// serves as entry point, creates new JNIImageFrame
public static void main( String[] args )
{
    JNIImageFrame imageFrame = new JNIImageFrame();
}

// constructs new JNIImageFrame containing JNIPanel
public JNIImageFrame()
{
    super( "Deitel Image Processing" );
    getContentPane().setLayout( new FlowLayout() );

    // adds new JNIPanel and defines image to process
    getContentPane().add( new JScrollPane( new JNIPanel( "advjhtp1_small.jpg" , new BorderLayout() ) ) );

    addWindowListener( new WindowAdapter() {
        public void windowClosing( WindowEvent event ) {
            dispose();
            System.exit( 0 );
        }
    } );
    setSize( 670, 500 );
    setVisible( true );
}

Fig. G.26 JNIImageFrame serves as an entry point for the application (part 2 of 2).
In this appendix, we presented the most common JNI techniques and functions. We have covered the fundamentals of interaction between Java and C++, including objects, arrays and exceptions. For additional JNI resources consult Section G.6.

G.6 Internet and World Wide Web Resources

www.java.sun.com/docs/books/tutorial/native1.1/index.html
This tutorial covers the fundamentals of JNI.

www.jguru.com/faq/subtopic.jsp?topicID=472
This site contains well over 100 frequently asked questions (FAQs) and answers.

www.acm.org/crossroads/xrds4-2/jni.html
This site is a JNI reference.

www.symbolstone.org/technology/java/index.html
A JNI book available for download. It includes detailed JNI information and tips on using JNI with applets.

SUMMARY

• The Java Native Interface (JNI) allows programmers to access existing applications written in other languages.
• JNI can be useful in time-critical applications—programmers can write a piece of the application in native code for performance reasons.
• Communication between Java and native code is slow, large amounts of communication will significantly degrade performance.
• Enabling a Java program to access native code requires a compiled wrapper class, a javah generated header and a library.
• The native keyword indicates that the method is implemented in a native library, not in Java.
• You must run javah on each Java class that contains a native method. Do not edit these headers—if you make a change to the wrapper class, run javah to generate appropriate header files again.
• jni.h is the header that contains all the JNI helper functions required for C++ to interact with Java.
• A signature is the argument and the return types of the function.
• #defines JNIEXPORT and JNICALL each help handle system specific JNI interactions with C++.
• JNIEnv is a pointer to a table of functions that assist the native code’s interaction with Java; it is always passed as the first argument to a JNI function.
• A JNIEnv* pointer named env would be accessed as env->functionName(args) in C++. In C, it would be accessed as (*env)->functionName(env, args).
• The second argument sent to a native function is a reference to the Java object that called it.
• The third through the final arguments sent to a native function are the arguments as they would be sent from Java code.
• Java Primitives and objects do not directly map to those used in C/C++.
• Java objects are passed as jobject.
• Strings are passed as jstring.
• Primitives are passed as \texttt{jtype}.
• Conversion between C/C++ and JNI primitive types is transparent to the programmer and may be done with simple assignment.
• Java uses Unicode to represent its characters, however C++ can only read the subset UTF-8 format.
• \texttt{GetStringUTFChars} converts a Unicode string to UTF-8 string.
• Releasing the string requires a call to \texttt{ReleaseStringUTFChars}, and must be done to avoid memory leaks.
• \texttt{System.loadLibrary} searches your library paths for a library with the given name and loads it into the Java JVM.
• Generally the library should be loaded in a \texttt{static} block. This ensures that the library is loaded before any native methods are called.
• If a native call is made before the library is loaded, the program will exit with an \texttt{UnsatisfiedLinkError}.
• \texttt{GetObjectClass} returns the \texttt{jclass} of the given object.
• A method ID is a unique ID associated with each method of a class, it may be obtained through the \texttt{GetMethodID} JNI function.
• Method signatures must be in the format: \texttt{(argument types) return type}.
• Primitive signatures are usually specified by their uppercase first letter e.g., \texttt{D} for \texttt{double}, \texttt{I} for \texttt{int} and \texttt{V} for \texttt{void}. Exceptions to this rule are \texttt{J} for \texttt{long} and \texttt{Z} for \texttt{boolean}.
• Object signatures must be prefixed by \texttt{L} followed by the class, including packages, and terminated by a semicolon, e.g., \texttt{Ljava/lang/String;} specifies the \texttt{String} type.
• Including a open bracket (\texttt{[}) before a signature indicates an array, \texttt{([D)V} represents a method that requires an array of \texttt{doubles} and returns \texttt{void}.
• Methods can be called with \texttt{CalltypeMethod}, where \texttt{type} represents the return type.
• JNI method \texttt{FindClass} returns a \texttt{jclass} reference to any fully qualified class name. If you have no instance of an object, \texttt{FindClass} must be called to retrieve the \texttt{jclass}.
• \texttt{<init>} is the method name used in JNI to signify a constructor.
• JNI function \texttt{NewObject} creates a new object of the given class.
• The standard JNI procedure to get a method or field is to obtain the \texttt{jclass}, get an ID for the member and access the member.
• \texttt{GetFieldID} obtains the field ID of the variable you wish to access.
• \texttt{GettypeField} returns the member contained within the given object at the \texttt{jfieldID} as type \texttt{jtype}.
• \texttt{GetStaticFieldID} obtains the \texttt{jfieldID} of a static member variable.
• Retrieving the field uses \texttt{GetStatictypeField}, and uses a \texttt{jclass} as the first argument, not the \texttt{jobject} used for non-static members.
• Converting arrays from Java to native code is complex because, as opposed to native languages, elements in a Java array are not always placed together in contiguous memory.
• The JNI function \texttt{NewObjectArray} creates a new array of the given object. \texttt{NewtypeArray} creates a new array of the specified primitive type.
• To obtain access to individual elements of a primitive array within native code requires a call to \texttt{GetPrimitiveTypeArrayElements}.
• \texttt{ReleaseTypeArrayElements} causes JNI to release the memory and update the original array. The argument \texttt{JNI_ABORT} frees the memory, but will not update the changes for Java, and argument \texttt{JNI_COMMIT} will update Java, but not free the array.
• **SetObjectArrayElements** and **GetObjectArrayElements** each provide simple access to an object array.

• There is no need to release object arrays. They are never copied from Java, thus Java’s built-in garbage collector will destroy them if they are unused.

• Handling exceptions natively can be essential to integrating native code with Java. JNI allows **native** methods to catch exceptions themselves, or to throw exceptions back to Java.

• JNI function **ThrowNew** signals Java to create a new exception of that type, with the given string as a message.

• Function **ExceptionOccurred** returns **JNI_NULL** if an exception was not thrown.

• JNI function **IsInstanceOf** takes an object and a class and returns **JNI_TRUE** if the object is an instance of the class. Accordingly, you may use **IsSameObject** to determine if two references are actually the same object.

• Call JNI function **Clear** to remove the error flag from JNI.

---

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>ANSI</th>
<th>JNI helper function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>jni.h</td>
</tr>
<tr>
<td>C++</td>
<td>JNI_ABORT</td>
</tr>
<tr>
<td>C++ library</td>
<td>JNI_COMMIT</td>
</tr>
<tr>
<td><code>CallStaticMethod</code> JNI function</td>
<td>JNI_FALSE</td>
</tr>
<tr>
<td><code>CallMethod</code> JNI function</td>
<td>JNI_TRUE</td>
</tr>
<tr>
<td>char *</td>
<td>JNI_CALLBACK</td>
</tr>
<tr>
<td><code>ExceptionOccurred</code> JNI function</td>
<td>JNIEnv variable</td>
</tr>
<tr>
<td>FatalError JNI function</td>
<td>JNIEXPORT</td>
</tr>
<tr>
<td>field ID</td>
<td>JNI-style header file</td>
</tr>
<tr>
<td><code>FindClass</code> JNI function</td>
<td>jobject</td>
</tr>
<tr>
<td><code>GetPrimitiveTypeArrayElement</code></td>
<td>jstring</td>
</tr>
<tr>
<td><code>GetFieldID</code> JNI function</td>
<td>jthrowable</td>
</tr>
<tr>
<td><code>GetMethodID</code> JNI function</td>
<td>library</td>
</tr>
<tr>
<td><code>GetObjectArrayElements</code> JNI function</td>
<td>method ID</td>
</tr>
<tr>
<td><code>GetStaticFieldID</code> JNI function</td>
<td>native</td>
</tr>
<tr>
<td><code>GetStaticMethodID</code> JNI function</td>
<td>native keyword</td>
</tr>
<tr>
<td><code>GetStaticTypeField</code> JNI function</td>
<td>native library</td>
</tr>
<tr>
<td><code>GetStaticTypeMethod</code> JNI function</td>
<td>JNI function</td>
</tr>
<tr>
<td><code>GetStringUTFChars</code> JNI function</td>
<td><code>NewObject</code></td>
</tr>
<tr>
<td><code>GetUTFChars</code> JNI function</td>
<td><code>NewObjectArray</code></td>
</tr>
<tr>
<td><code>IsInstanceOf</code> JNI function</td>
<td><code>NewtypeArray</code></td>
</tr>
<tr>
<td><code>IsSameObject</code> JNI function</td>
<td><code>OutOfMemoryError</code></td>
</tr>
<tr>
<td>Java Native Interface</td>
<td><code>ReleaseStringUTFChars</code></td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>SetObjectArrayElement</code></td>
</tr>
<tr>
<td><code>jboolean</code></td>
<td><code>(JNIEnv *)</code></td>
</tr>
<tr>
<td><code>jclass</code></td>
<td>signature</td>
</tr>
<tr>
<td><code>jdouble</code></td>
<td><code>System.loadLibrary</code></td>
</tr>
<tr>
<td><code>jint</code></td>
<td><code>ThrowNew</code></td>
</tr>
<tr>
<td><code>jintArray</code></td>
<td>Unicode</td>
</tr>
<tr>
<td>JNI</td>
<td><code>UnsatisfiedLinkError</code></td>
</tr>
<tr>
<td></td>
<td>UTF-8</td>
</tr>
<tr>
<td></td>
<td>wrapper class</td>
</tr>
</tbody>
</table>
SELF-REVIEW EXERCISES

G.1 Fill in the blanks for each of the following:
   a) Code that can be compiled to a binary format is also known as ________ code.
   b) ________ is the tool used to create C headers for JNI.
   c) To convert Unicode strings to UTF-8 strings the ________ function of the JNIEnv pointer may be used.
   d) Accessing a non-static method from native code requires three steps: ________, ________, and ________.
   e) The CallStatic type Method differs from the Call type Method because it requires an argument of type ________ while Call type Method uses an argument of type ________.

G.2 State whether each of the following is true or false. If false explain your answers.
   a) Java is a native language.
   b) Type JNIEnv does not refer to an object.
   c) Java primitives can be converted to C++ primitives through assignment.
   d) JNI always increases program performance.
   e) Java objects may only be created from a Java program and passed to the native function.

ANSWERS TO SELF-REVIEW EXERCISES

G.1 a) native. b) javah. c) GetStringUTFChars. d) obtain jclass, obtain method ID, call the method. e) jclass, jobject.

G.2 a) False, Java is not compiled to a binary format. b) True. c) True. d) False, JNI often reduces performance due to the overhead associated with JNI calls. e) False, native code can create new Java objects.

EXERCISES

G.3 Explain why Java arrays do not directly map to C++ arrays. Explain how JNI deals with this complication, and contrast between primitive and object arrays.

G.4 Define the term “method signatures” and explain why JNI needs them.

G.5 What should you be careful with after throwing an exception in native code? Why?

G.6 Write native code to obtain the method ID of the constructor for class Integer that takes an int as an argument. Assume you already have the class stored in jclass integerClass.

G.7 Write the native code required to call the Java method isEmpty of class Vector. Assume you have an instance of a Vector assigned to jobject myVector.

G.8 Create a wrapper class MathWrapper that contains three native functions cSin, cCos, and cLN, each taking a double for an argument and returning a double. The wrapper class should explicitly declare that the cLN function throws an Exception. Create library MathLibrary. It should implement the header created from running javah on MathWrapper and contain three JNI-style functions that encapsulate the use of sin, cos, and ln of C++ header math.h. If the argument to ln is less than or equal to zero, throw an exception, and have Java handle the exception. Finally, write a Java class MathMain that calls native methods for the math operations in class MathWrapper. It should load the native library, MathLibrary, and test each function with arguments entered by the user. cLN must catch an Exception and print its message.
BIBLIOGRAPHY
Career Opportunities

**Objectives**

- To explore the various online career services.
- To examine the advantages and disadvantages of posting and finding jobs online.
- To review the major online career services Web sites available to job seekers.
- To explore the various online services available to employers seeking to build their workforces.

*What is the city but the people?*
William Shakespeare

*A great city is that which has the greatest men and women,*
*If it be a few ragged huts it is still the greatest city in the whole world.*
Walt Whitman

*To understand the true quality of people, you must look into their minds, and examine their pursuits and aversions.*
Marcus Aurelius

*The soul is made for action, and cannot rest till it be employed. Idleness is its rust. Unless it will up and think and taste and see, all is in vain.*
Thomas Traherne
There are approximately 40,000 career-advancement services on the Internet today.1 These services include large, comprehensive job sites, such as Monster.com (see the upcoming Monster.com feature), as well as interest-specific job sites such as JustJava-Jobs.com. Companies can reduce the amount of time spent searching for qualified employees by building a recruiting feature on their sites or establishing an account with a career site. This results in a larger pool of qualified applicants, as online services can automatically select and reject resumes based on user-designated criteria. Online interviews, testing services and other resources also expedite the recruiting process.

Applying for a position online is a relatively new method of exploring career opportunities. Online recruiting services streamline the process and allow job seekers to concentrate their energies in careers that are of interest to them. Job seekers can explore opportunities according to geographic location, position, salary or benefits packages.

Job seekers can learn how to write a resume and cover letter, post them online and search through job listings to find the jobs that best suit their needs. Entry-level positions, or positions commonly sought by individuals who are entering a specific field or the job market for the first time; contracting positions; executive-level positions and middle-management-level positions are all available on the Web.
Job seekers will find a number of time-saving features when searching for a job online. These include storing and distributing resumes digitally, e-mail notification of possible positions, salary and relocation calculators, job coaches, self-assessment tools and information on continuing education.

In this appendix, we explore online career services from the employer and employee’s perspective. We suggest sites on which applications can be submitted, jobs can be searched for and applicants can be reviewed. We also review services that build recruiting pages directly into an e-business.

### H.2 Resources for the Job Seeker

Finding a job online can greatly reduce the amount of time spent applying for a position. Instead of searching through newspapers and mailing resumes, job seekers can request a specific position in a specific industry through a search engine. Some sites allow job seekers to setup intelligent agents to find jobs that meet their requirements. Intelligent agents are programs that search and arrange large amounts of data, and report answers based on that data. When the agent finds a potential match, it sends it to the job seeker’s inbox. Resumes can be stored digitally, customized quickly to meet job requirements and e-mailed instantaneously. Potential candidates can also learn more about a company by visiting its Web site. Most employment sites are free to job seekers. These sites typically generate their revenues by charging employers for posting job opportunities and by selling advertising space on their Web pages (see the Monster.com feature).

Career services, such as FlipDog.com, search a list of employer job sites to find positions. By searching links to employer Web sites, FlipDog.com is able to identify positions from companies of all sizes. This feature enables job seekers to find jobs that employers may not have posted outside the corporation’s Web site.

---

**Monster.com**

Super Bowl ads and effective marketing have made Monster.com one of the most recognizable online brands (see Fig. B.1). In fact, in the 24 hours following Super Bowl XXXIV, 5 million job searches occurred on Monster.com.² The site allows people looking for jobs to post their resumes, search job listings, read advice and information about the job-search process and take proactive steps to improve their careers. These services are free to job seekers. Employers can post job listings, search resume databases and become featured employers.

Posting a resume at Monster.com is simple and free. Monster.com has a resume builder that allows users to post a resume to its site in 15–30 minutes. Each user can store up to 5 resumes and cover letters on the Monster.com server. Some companies offer their employment applications directly through the Monster.com site. Monster.com has job postings in every state and all major categories. Users can limit access to their personal identification information. As one of the leading recruiting sites on the Web, Monster.com is a good place to begin a job search or to find out more about the search process.
Job seekers can visit FlipDog.com and choose, by state, the area in which they are looking for a position. Applicants can also conduct worldwide searches. After a user selects a region, FlipDog.com requests the user to specify a job category containing several specific positions. The user’s choice causes a list of local employers to appear. The user can choose a specific employer or request that FlipDog.com search the employment databases for jobs offered by all employers (see Fig. B.2).

Other services, such as employment networks, also help job seekers in their search. Sites such as Vault.com (see the Vault.com feature) and WetFeet.com allow job seekers to post questions about employers and positions in designated chat rooms and on bulletin boards.

**H.3 Online Opportunities for Employers**

Recruiting on the Internet provides several benefits over traditional recruiting. For example, Web recruiting reaches a much larger audience than posting an advertisement in a local newspaper. Given the breadth of the services provided by most online career services Web sites, the cost of posting online can be considerably less expensive than posting positions through traditional means. Even newspapers, which depend greatly on career opportunity advertising, are starting online career sites.³
Vault.com: Finding the Right Job on the Web

Vault.com allows potential employees to seek out additional, third-party information for over 3000 companies. By visiting the Insider Research page, Web users have access to a profile on the company of their choice, as long as it exists in Vault.com’s database. In addition to Vault.com’s profile, there is a link to additional commentary by company employees. Most often anonymous, these messages can provide prospective employees with potentially valuable decision-making information. However, users must consider the integrity of the source. For example, a disgruntled employee may leave a posting that is not an accurate representation of the corporate culture of his or her company.

The Vault.com Electronic Watercooler™ is a message board that allows visitors to post stories, questions and concerns and to advise employees and job seekers. In addition, the site provides e-newsletters and feature stories designed to help job seekers in their search. Individuals seeking information on business, law and graduate schools can also find information on Vault.com.

Job-posting and career-advancement services for the job seeker are featured on Vault.com. These services include VaultMatch, a career service that e-mails job postings as requested, and Salary Wizard™, which helps job seekers determine the salary they are worth. Online guides with advice for fulfilling career ambitions are also available.
According to Forrester Research, 33 percent of today’s average company’s hiring budget goes toward online career services, while the remaining 66 percent is used toward traditional recruiting mechanisms. Online use is expected to increase to 42 percent by 2004, while traditional mechanisms may be reduced to 10 percent.\(^5\)

Generally, jobs posted online are viewed by a larger number of job seekers than jobs posted through traditional means. However, it is important not to overlook the benefits of combining online efforts with human-to-human interaction. There are many job seekers who are not yet comfortable with the process of finding a job online. Often, online recruiting is used as a means of freeing up a recruiter’s time for the interviewing process and final selection.

Cisco Systems cites a 39 percent reduction in cost-per-hire expenses, and a 60 percent reduction in the time spent hiring.\(^6\)

### H.3.1 Posting Jobs Online

When searching for job candidates online, there are many things employers need to consider. The Internet is a valuable tool for recruiting, but one that takes careful planning to acquire the best results. It provides a good supplementary tool, but should not be considered the complete solution for filling positions. Web sites, such as WebHire (www.webhire.com), enhance a company’s online employment search (see the WebHire feature).

There are a variety of sites that allow employers to post jobs online. Some of these sites require a fee, which generally runs between $100–200. Postings typically remain on the Web site for 30–60 days. Employers should be careful to post to sites that are most likely to be visited by eligible candidates. As we discovered in the previous section, there are a variety of online career services focused on specific industries, and many of the larger, more comprehensive sites have categorized their databases by job category.

When designing a posting, the recruiter should consider the vast number of postings already on the Web. Defining what makes the job position unique, including information such as benefits and salary, might convince a qualified candidate to further investigate the position (see Fig. B.3).\(^7\)

HotJobs.com career postings are cross-listed on a variety of other sites, thus increasing the number of potential employees who see the job listings. Like Monster.com and jobfind.com.hotjobs.com requires a fee per listing. Employers also have the option of becoming HotJob.com members. Employers can gain access to HotJob’s Private Label Job Boards (private corporate employment sites), online recruiting technology and online career fairs.
**WebHire™**

Designed specifically for recruiters and employers, WebHire is a multifaceted service that provides employers with *end-to-end recruiting solutions*. The service offers job-posting services as well as candidate searches. The most comprehensive of the services, **WebHire™ Enterprise**, locates and ranks candidates found through resume-scanning mechanisms. Clients will also receive a report indicating the best resources for their search. Other services available through the **WebHire™ Employment Services Network** include preemployment screening, tools for assessing employees’ skill levels and information on compensation packages. An employment law advisor helps organizations design interview questions.

**WebHire™ Agent** is an intelligent agent that searches for qualified applicants based on job specifications. When WebHire Agent identifies a potential candidate, an e-mail is automatically sent to the candidate to generate interest. WebHire Agent then ranks applicants according to the skills information it gains from the Web search; the information is stored so that new applicants are distinguished from those who have already received an e-mail from the site.

**Yahoo!® Resumes**, a feature of WebHire, allows recruiters to find potential employees by typing in keywords on the Yahoo! Resumes search engine. Employers can purchase a year’s membership to the recruiting solution for a flat fee; there are no per-use charges.

---

**Job Seeker’s Criteria**

- Position (responsibilities)
- Salary
- Location
- Benefits (health, dental, stock options)
- Advancement
- Time Commitment
- Training Opportunities
- Tuition Reimbursement
- Corporate Culture

---

**Fig. H.3** List of a job seeker’s criteria.

Boston Herald *Job Find* ([www.jobfind.com](http://www.jobfind.com)) also charges employers to post on its site. The initial fee entitles the employer to post up to three listings. Employers have no limitations on the length of their postings.

Other Web sites providing employers with employee recruitment services include **CareerPath.com**, America’s Job Bank ([www.ajb.dni.us/employer](http://www.ajb.dni.us/employer)), **CareerWeb** ([www.cweb.com](http://www.cweb.com)), **Jobs.com** and **Career.com**.
H.3.2 Problems with Recruiting on the Web

The large number of applicants presents a challenge to both job seekers and employers. On many recruitment sites, matching resumes to positions is conducted by *resume-filtering software*. The software scans a pool of resumes for keywords that match the job description. While this software increases the number of resumes that receive attention, it is not a foolproof system. For example, the resume-filtering software might overlook someone with similar skills to those listed in the job description, or someone whose abilities would enable them to learn the skills required for the position. Digital transmissions can also create problems because certain software platforms are not always acceptable by the recruiting software. This sometimes results in an unformatted transmission, or a failed transmission.

A lack of confidentiality is another disadvantage of online career services. In many cases, a job candidate will want to search for job opportunities anonymously. This reduces the possibility of offending the candidate’s current employer. Posting a resume on the Web increases the likelihood that the candidate’s employer might come across it when recruiting new employees. The traditional method of mailing resumes and cover letters to potential employers does not impose the same risk.

According to recent studies, the number of individuals researching employment positions through traditional means, such as referrals, newspapers and temporary agencies, far outweighs the number of job seekers researching positions through the Internet. Optimists feel, however, that this disparity is largely due to the early stages of e-business development. Given time, online career services will become more refined in their posting and searching capabilities, decreasing the amount of time it takes for a job seeker to find jobs and employers to fill positions.

H.3.3 Diversity in the Workplace

Every workplace inevitably develops its own culture. Responsibilities, schedules, deadlines and projects all contribute to a working environment. Perhaps the most defining elements of a *corporate culture* are the employees. For example, if all employees were to have the same skills and the same ideas, the workplace would lack diversity. It might also lack creativity and enthusiasm. One way to increase the dynamics of an organization is to employ people of all backgrounds and cultures.

The Internet hosts demographic-specific sites for employers seeking to increase diversity in the workplace. By recruiting people from different backgrounds, new ideas and perspectives are brought forth, helping businesses meet the needs of a larger, more diverse target audience.

Blackvoices.com and hirediversity.com are demographic-specific Web sites. BlackVoices™, which functions primarily as a portal (a site offering news, sports and weather information, as well as the ability to search the Web), features job searching capabilities and the ability for prospective employees to post resumes. HireDiversity is divided into several categories, including opportunities for African Americans, Hispanics and women. Other online recruiting services place banner advertisements on ethnic Web sites for companies seeking diverse workforces.

The Diversity Directory ([www.mindexchange.com](http://www.mindexchange.com)) offers international career-searching capabilities. Users selecting the Diversity site can find job opportunities, information and additional resources to help them in their career search. The site can be searched
according to demographics (African American, Hispanic, alternative lifestyle, etc.) or by subject (employer, position, etc.) via hundreds of links. Featured sites include **BilingualJobs.com, Latin World** and **American Society for Female Entrepreneurs**.

Many sites have sections dedicated to job seekers with disabilities. In addition to providing job-searching capabilities, these sites include additional resources, such as equal opportunity documents and message boards. The **National Business and Disability Council (NBDC)** provides employers with integration and accessibility information for employing people with disabilities, and the site also lists opportunities for job seekers.

### H.4 Recruiting Services

There are many services on the Internet that help employers match individuals to positions. The time saved by conducting preliminary searches on the Internet can be dedicated to interviewing qualified candidates and making the best matches possible.

Advantage Hiring, Inc. ([www.advantagehiring.com](http://www.advantagehiring.com)) provides employers with a resume-screening service. When a prospective employee submits a resume for a particular position, Advantage Hiring, Inc. presents **Net-Interview™**, a small questionnaire to supplement the information presented on the resume. The site also offers **SiteBuilder**, a service that helps employers build an employee recruitment site. An online demonstration can be found at [www.advantagehiring.com](http://www.advantagehiring.com). The demonstration walks the user through the Net-Interview software, as well as a number of other services offered by Advantage Hiring (see Fig. B.4).

**Recruitsoft.com** is an application service provider (ASP) that offers companies recruiting software on a **pay-per-hire** basis (Recruitsoft receives a commission on hires made via its service). **Recruiter WebTop™** is the company’s online recruiting software. It includes features such as Web-site hosting, an employee-referral program, skill-based resume screening, applicant-tracking capabilities and job-board posting capabilities. A demonstration of Recruiter WebTop’s **Corporate Recruiting Solutions** can be found at [www.recruitsoft.com/process](http://www.recruitsoft.com/process). The demonstration shows how recruiting solutions find and rank potential candidates. More information about Recruitsoft’s solution can be viewed in a **QuickTime** media player demonstration, found at [www.recruitsoft.com/corpoVideo](http://www.recruitsoft.com/corpoVideo).

**Peoplescape.com** is an online service that helps employers recruit employees and maintain a positive work environment once the employee has been hired. In addition to searches for potential candidates, Peoplescape offers **PayCheck™, LegalCheck™ and PeopleCheck™**. These services help to ensure that compensation offers are adequate, legal guidelines are met and candidates have provided accurate information on their resumes and during the hiring process. For job seekers, Peoplescape offers searching capabilities, insights to career transitions, a job compensation calculator that takes benefits and bonuses into consideration when exploring a new job possibility and a series of regularly posted articles relevant to the job search.¹¹

To further assist companies in their recruiting process, Web sites such as **Refer.com** reward visitors for successful job referrals. Highly sought-after positions can earn thousands of dollars. If a user refers a friend or a family member and he or she is hired, the user receives a commission.

Other online recruiting services include **SkillsVillage.com, Hire.com, MorganWorks.com** and **Futurestep.com™**.
H.4.1 Testing Potential Employees Online

The Internet also provides employers with a cost-effective means of testing their prospective employees in such categories as decision making, problem solving and personality. Services such as eTest help to reduce the cost of in-house testing and to make the interview process more effective. Test results, given in paragraph form, present employers with the interested individual’s strengths and weaknesses. Based on these results, the report suggests interview methods, such as asking open-ended questions, which are questions that require more than a “yes” or “no” response. Sample reports and a free-trial test can be found at www.etest.net.

Employers and job seekers can also find career placement exercises at www.advisorteam.net/AT/User/kcs.asp. Some of these services require a fee. The tests ask several questions regarding the individual’s interests and working style. Results help candidates determine the best career for their skills and interests.
H.5 Career Sites

Online career sites can be comprehensive or industry specific. In this section, we explore a variety of sites on the Web that accommodate the needs of both the job seeker and the employer. We review sites offering technical positions, free-lancing opportunities and contracting positions.

H.5.1 Comprehensive Career Sites

As mentioned previously, there are many sites on the Web that provide job seekers with career opportunities in multiple fields. **Monster.com** is the largest of these sites, attracting the greatest number of unique visitors per month. Other popular online recruiting sites include **JobsOnline.com**, **HotJobs.com**, **www.jobtrak.com** and **Headhunter.net**.

Searching for a job online can be conducted in a few steps. For example, during an initial visit to **JobsOnline.com**, a user is required to fill out a registration form. The form requests basic information, such as name, address and area of interest. After registering, members can search through job postings according to such criteria as job category, location and the number of days the job has been posted. Contact information is provided for additional communication. Registered members are offered access to XDrive™ (**www.xdrive.com**), which provides 25 MB of storage space for resumes, cover letters and additional communication. Stored files can be shared through any Web browser or Wireless Application Protocol (WAP)-enabled device. **Driveway.com** offers a similar service, allowing individuals to store, share and organize job search files online. An online demonstration of the service can be found at **www.driveway.com**. The animated demo walks the user through the features offered by the service. **Driveway.com** offers 100 MB of space, and the service is free. Other sites, such as Cruel World (see the Cruel World feature), allow users to store and send their resumes directly to employers.

**Cruel World**

Cruel World is a free, online career advancement service for job seekers. After becoming a registered member, your information is matched with available positions in the Cruel World database. When an available job matches your criteria, **JobCast®**, a feature of Cruel World, sends an e-mail alerting you of the available position. If you are interested, you can send your resume to the employer that posted the position, customized to the job’s requirements. If you do not wish to continue your search, you can simply send a negative response via e-mail.

The client list, or the list of companies seeking new employees through Cruel World, can be viewed at **www.cruelworld.com/corporate/aboutus.asp** (Fig. B.5). Additional features on the site include hints for salary negotiation; a self-assessment link to **CareerLeader.com**, where, for a small fee, members can reassess their career goals under the advisement of career counselors and a relocation calculator for job seekers who are considering changing location.

Employers seeking to hire new talent can post opportunities through Cruel World. posting positions requires a fee. A demonstration of the service can be viewed at **www.cruelworld.com/clients/quicktour1.asp**. The demonstration is a three-step slide of JobCast.
H.5.2 Technical Positions

Technical positions are becoming widely available as the Internet grows more pervasive. Limited job loyalty and high turnover rates in technical positions allow job seekers to find jobs that best suit their needs and skills. Employers are required to rehire continuously to keep positions filled and productivity levels high. The amount of time for an employer to fill a technical position can be greatly reduced by using an industry-specific site. Career sites designed for individuals seeking technical positions are among the most popular online career sites. In this section, we review several sites that offer recruiting and hiring opportunities for technical positions.

**e-Fact H.3**

It costs a company 25 percent more to hire a new technical employee than it does to pay an already employed individual’s salary.14

**Dice.com (www.dice.com)** is a recruiting Web site that focuses on technical fields. Company fees are based on the number of jobs the company posts and the frequency
with which the postings are updated. Job seekers can post their resumes and search the job database for free. JustComputerJobs.com directs job seekers toward 39 specific computer technologies for their job search. Language-specific sites include JustJavaJobs.com, JustCJobs.com and JustPerlJobs.com. Hardware, software and communications technology sites are also available. Other technology recruiting sites include HireAbility.com, Bid4Geeks.com, HotDispatch.com and www.cmpnet.com/careerdirect.

H.5.3 Wireless Positions

The wireless industry is developing rapidly. According to WirelessResumes.com, the number of wireless professionals is 328,000. This number is expected to increase 40 percent each year for the next five years. To accommodate this growth, and the parallel demand for professionals, WirelessResumes.com has created an online career site specifically for the purpose of filling wireless jobs (see the WirelessResumes.com feature).

The Caradyne Group (www.pcsjobs.com), an executive search firm, connects job seekers to employers in the wireless technology field. Interested job seekers must first fill out a “Profile Questionnaire.” This information is then entered into The Caradyne Group’s database and is automatically matched to an open position in the job seeker’s field of expertise. If there are no open positions, a qualified consultant from The Caradyne Group will contact the job seeker for further interview and discussion. Jobs4wireless.com also provides job seekers with employment opportunities in the wireless industry.

H.5.4 Contracting Online

The Internet also serves as a forum for job seekers to find employment on a project-by-project basis. Online contracting services allow businesses to post positions for which they wish to hire outside resources, and individuals can identify projects that best suit their interests, schedules and skills.

e-Fact H.4

Approximately six percent of America’s workforce falls into the category of independent contractor.15

WirelessResumes.com: Filling Wireless Positions

WirelessResumes.com is an online career site focused specifically on matching wireless professionals with careers in the industry. This narrow focus enables businesses to locate new employees quickly—reducing the time and expense attached to traditional recruiting methods. Similarly, candidates can limit their searches to precisely the job category of interest. Wireless carriers, device manufacturers, WAP and Bluetooth developers, e-commerce companies and application service providers (ASPs) are among those represented on the site.

In addition to searching for jobs and posting a resume, WirelessResumes.com provides job seekers with resume writing tips, interviewing techniques, relocation tools and assistance in obtaining a Visa or the completion of other necessary paperwork. Employers can use the site to search candidates and post job opportunities.
Guru.com (www.guru.com) is a recruiting site for contract employees. Independent contractors, private consultants and trainers use guru.com to find short-term and long-term contract assignments. Tips, articles and advice are available for contractors who wish to learn more about their industry. Other sections of the site teach users how to manage their businesses, buy the best equipment and deal with legal issues. Guru.com includes an online store where contractors can buy products associated with small-business management, such as printing services and office supplies. Companies wishing to hire contractors must register with guru.com, but individuals seeking contract assignments do not.

Monster.com’s Talent Market™ offers online auction-style career services to free agents. Interested users design a profile, listing their qualifications. After establishing a profile, free agents “Go Live” to start the bidding on their services. The bidding lasts for five days during which users can view the incoming bids. At the close of five days, the user can choose the job of his or her choice. The service is free for users, and bidding employers pay a commission on completed transactions.

eLance.com is another site where individuals can find contracting work. Interested applicants can search eLance’s database by category, including business, finance and marketing (see Fig. B.6). These projects, or requests for proposals (RFPs), are posted by companies worldwide. When users find projects for which they feel qualified, they submit bids on the projects. Bids must contain a user’s required payment, a statement detailing the user’s skills and a feedback rating drawn from other projects on which the user has worked. If a user’s bid is accepted, the user is given the project, and the work is conducted over eLance’s file-sharing system, enabling both the contractor and the employer to contact one another quickly and easily. For an online demonstration, visit www.elance.com and click on the demonstration icon.

FreeAgent (www.freeagent.com) is another site designed for contracting projects. Candidates create an e.portfolio that provides an introductory “snapshot” of their skills, a biography, a list of their experience and references. The interview section of the portfolio lists questions and the applicant’s answers. Examples of e.portfolios can be found at www.freeagent.com/splash/models.asp. Free Agent’s e.office offers a benefits package to outside contractors, including health insurance, a retirement plan and reimbursement for business-related expenses.

Other Web sites that provide contractors with projects and information include eWork® Exchange (www.ework.com), MBAFreeAgent.com, Aquent.com and WorkingSolo.com.

H.5.5 Executive Positions

Next, we discuss the advantages and disadvantages of finding an executive position online. Executive career advancement sites usually include many of the features found on comprehensive job-search sites. Searching for an executive position online differs from finding an entry-level position online. The Internet allows individuals to continually survey the job market. However, candidates for executive-level positions must exercise a higher level of caution when determining who is able to view their resume. Applying for an executive position online is an extensive process. As a result of the high level of scrutiny passed on a candidate during the hiring process, the initial criteria presented by an executive level candidate often are more specific than the criteria presented by the first-time job seeker. Executive positions often are difficult to fill, due to the high demands and large amount of experience required for the jobs.
SixFigureJobs (www.sixfigurejobs.com) is a recruitment site designed for experienced executives. Resume posting and job searching is free to job seekers. Other sites, including www.execunet.com, Monster.com’s ChiefMonster™ (www.chiefmonster.com) and www.nationjob.com are designed for helping executives find positions.

**H.5.6 Students and Young Professionals**

The Internet provides students and young professionals with tools to get them started in the job market. Individuals still in school and seeking internships, individuals who are just graduating and individuals who have been in the workforce for a few years make up the target market. Additional tools specifically designed for this demographic (a population defined by a specific characteristic) are available. For example, journals kept by previous interns provide prospective interns with information regarding what to look for in an internship, what to expect and what to avoid. Many sites will provide information to lead young professionals in the right direction, such as matching positions to their college or university major.

**Experience.com** is a career services Web site geared toward the younger population. Members can search for positions according to specific criteria, such as geo-
Appendix H Career Opportunities 1753

graphic location, job category, keywords, commitment (i.e. full time, part time, internship), amount of vacation and amount of travel time. After applicants register, they can send their resumes directly to the companies posted on the site. In addition to the resume, candidates provide a personal statement, a list of applicable skills and their language proficiency. Registered members also receive access to the site’s Job Agent. Up to three Job Agents can be used by each member. The agents search for available positions, based on the criteria posted by the member. If a match is made, the site contacts the candidate via e-mail.16,17

**Internshipprograms.com** helps students find internships. In addition to posting a resume and searching for an internship, students can use the relocation calculator to compare the cost of living in different regions. Tips on building resumes and writing essays are provided. The City Intern program provides travel, housing and entertainment guides to interns interviewing or accepting a position in an unfamiliar city, making them feel more at home in a new location.

In addition to its internship locators, undergraduate, graduate, law school, medical school and business school services, the Princeton Review’s Web site ([www.review.com](http://www.review.com)) offers career services to graduating students. While searching for a job, students and young professionals can also read through the site’s news reports or even increase their vocabulary by visiting the “word for the day.” Other career sites geared toward the younger population include [campuscareercenter.com](http://campuscareercenter.com), [brassrings-campus.com](http://brassrings-campus.com) and [collegegrads.com](http://collegegrads.com).

### H.5.7 Other Online Career Services

In addition to Web sites that help users find and post jobs online, there are a number of Web sites that offer features that will enhance searches, prepare users to search online, help applicants design resumes or help users calculate the cost of relocating.

**Salary.com** helps job seekers gauge their expected income, based on position, level of responsibility and years of experience. The search requires job category, ZIP code and specific job title. Based on this information, the site will return an estimated salary for an individual living in the specified area and employed in the position described. Estimates are returned based on the average level of income for the position.

In addition to helping applicants find employment, [www.careerpower.com](http://www.careerpower.com) provides individuals with tests that will help them realize their strengths, weaknesses, values, skills and personality traits. Based on the results, which can be up to 10–12 pages per test, users can best decide what job categories they are qualified for and what career choice will be best suited to their personal ambitions. The service is available for a fee.

InterviewSmart™ is another service offered through CareerPower that prepares job seekers of all levels for the interviewing process. The service can be downloaded for a minimal fee or can be used on the Web for free. Both versions are available at [www.careerpower.com/CareerPerfect/interviewing.htm#is.start.anchor](http://www.careerpower.com/CareerPerfect/interviewing.htm#is.start.anchor).

Additional services will help applicants find positions that meet their unique needs, or design their resumes to attract the attention of specific employers. [Dogfriendly.com](http://Dogfriendly.com), organized by geographic location, helps job seekers find opportunities that allow them to bring their pets to work, and [cooljobs.com](http://cooljobs.com) is a searchable database of unique job opportunities.
H.6 Internet and World Wide Web Resources

**Information Technology (IT) Career Sites**

www.dice.com
This is a recruiting Web site that focuses on the computer industry.

www.guru.com
This is a recruiting site for contract employees. Independent contractors, private consultants and trainers can use guru.com to find short-term and long-term work.

www.hallkinion.com
This is a Web recruiting service for individuals seeking IT positions.

www.techrepublic.com
This site provides employers and job seekers with recruiting capabilities and information regarding developing technology.

www.justcomputerjobs.com
This site serves as a portal with access to language-specific sites, including Java, Perl, C and C++.

www.bid4geeks.com
This career services site is geared toward the technical professional.

www.hotdispatch.com
This forum provides software developers with the opportunity to share projects, discuss code and ask questions.

www.techjobs.bizhosting.com/jobs.htm
This site directs job seekers to links of numerous technological careers listed by location, internet, type of field, etc.

**Career Sites**

www.careerbuilder.com
A network of career sites, including IT Careers, USA Today and MSN, CareerBuilder attracts 3 million unique job seekers per month. The site provides resume-builder and job-searching agents.

www.recruitek.com
This free site caters to jobs seekers, employers and contractors.

www.monster.com
This site, the largest of the online career sites, allows people looking for jobs to post their resumes, search job listings and read advice and information about the job-search process. It also provides a variety of recruitment services for employers.

www.jobsonline.com
Similar to Monster.com, this site provides opportunities for job seekers and employers.

www.hotjobs.com
This online recruiting site offers cross-listing possibilities on additional sites.

www.jobfind.com
This job site is an example of locally targeted job-search resources. JobFind.com targets the Boston area.

www.flipdog.com
This site allows online job candidates to search for career opportunities. It employs intelligent agents to scour the Web and return jobs matching the candidate’s request.
www.cooljobs.com
This site highlights unique job opportunities.

www.careerhighway.com
This site presents an opportunity for job seekers and employers to match up and register the career-specific information for which they are searching.

www.inetsupermall.com
This site aids job seekers in creating professional resumes and connecting with employers.

www.wirelessnetworksonline.com
This site helps connect job seekers to careers for which they are qualified.

www.careerweb.com
This site highlights featured employers and jobs and allows job seekers and employers to post and view resumes, respectively.

Executive Positions
www.sixfigurejobs.com
This is a recruitment site designed for experienced executives.

www.leadersonline.com
This career services Web site offers confidential job searches for mid-level professionals. Potential job matches are e-mailed to job candidates.

www.ecruitinginc.com
This site is designed to search for employees for executive positions.

Diversity
www.latpro.com
This site is designed for Spanish-speaking and Portuguese-speaking job seekers. In addition to providing resume-posting services, the site enables job seekers to receive matching positions via e-mail. Advice and information services are available.

www.blackvoices.com
This portal site hosts a career center designed to match African American job seekers with job opportunities.

www.hirediversity.com
In addition to services for searching for and posting positions, resume-building and updating services are also available on this site. The site targets a variety of demographics including African Americans, Asian Americans, people with disabilities, women and Latin Americans.

People with Disabilities
www.halftheplanet.com
This site represents people with disabilities. The site is large and includes many different resources and information services. A special section is dedicated to job seekers and employers.

www.wemedia.com
This site is designed to meet the needs of people with disabilities. It includes a section for job seekers and employers.

www.disabilities.com
This site provides users with a host of links to information resources on career opportunities.
www.rileyguide.com
This site includes a section with opportunities for people with disabilities, which can be viewed at www.dbm.com/jobguide/vets.html#abled.

www.mindexchange.com
The diversity section of this site provides users with several links to additional resources regarding people with disabilities and employment.

www.usdoj.gov/crt/ada/adahom1.htm
This is the Americans with Disabilities Act home page.

www.abanet.org/disability/home.html
This is the Web site for The Commission on Mental and Physical Disability Law.

janweb.icdi.wvu.edu
The Job Accommodation Web site offers consulting services to employers regarding integration of people with disabilities into the workplace.

General Resources

www.vault.com
This site provides potential employees with “insider information” on over 3000 companies. In addition, job seekers can search through available positions and post and answer questions on the message board.

www.wetfeet.com
Similar to vault.com, this site allows visitors to ask questions and receive “insider information” on companies that are hiring.

Free Services

www.sleuth.com
On this site job seekers can fill out a form that indicates their desired field of employment. Job Sleuth™ searches the Internet and returns potential matches to the user’s inbox. The service is free.

www.ajb.org
America’s Job Bank is an online recruiting service provided through the Department of Labor and the state employment service. Searching for and posting positions on the site are free.

www.xdrive.com
This free site provides members with 25 MB of storage space for housing documents related to a user’s job search. XDrive is able to communicate with all browser types and has wireless capabilities.

www.driveway.com
Similar to XDrive.com, this Web site provides users with 100 MB of storage space. Users can back up, share and organize information about various job searches. Driveway.com works on all platforms.

Special Interest

www.eharvest.com/careers/index.cfm
This Web site provides job seekers interested in agricultural positions with online career services.

www.opportunitynocs.org
This career services site is for both employers and job seekers interested in non-profit opportunities.

www.experience.com
This Web site is designed specifically for young professionals and students seeking full-time, part-time and internship positions.
www.internshipprograms.com
Students seeking internships can search job listings on this site. It also features City Intern, to help interns become acquainted with a new location.

www.brassringcampus.com
This site provides college grads and young professionals with less than five years of experience with job opportunities. Additional features help users buy cars or find apartments.

**Online Contracting**

www.ework.com
This online recruiting site matches outside contractors with companies needing project specialists. Other services provided through eWork include links to online training sites, benefits packages and payment services and online meeting and management resources.

www.elance.com
Similar to eWork.com, eLance matches outside contractors with projects.

www.freeagent.com
FreeAgent matches contractors with projects.

www.MBAFreeAgent.com
This site is designed to match MBAs with contracting opportunities.

www.aquent.com
This site provides access to technical contracting positions.

www.WorkingSolo.com
This site helps contractors begin their own projects.

**Recruiting Services**

www.advantagehiring.com
This site helps employers screen resumes.

www.etest.net
This site provides employers with testing services to assess the strengths and weaknesses of prospective employees. This information can be used for better hiring strategies.

www.hire.com
Hire.com’s eRecruiter is an application service provider that helps organizations streamline their Web-recruiting process.

www.futurestep.com
Executives can register confidentially at Futurestep.com to be considered for senior executive positions. The site connects registered individuals to positions. It also offers career management services.

www.webhire.com
This site provides employers with end-to-end recruiting solutions.

**Wireless Career Resources**

www.wirelessresumes.com
This site connects employers and job seekers with resumes that focus on jobs revolving around wireless technology.

www.msua.org/job.htm
This site contains links to numerous wireless job-seeking Web sites.
www.jobs4wireless.com
This site searches for jobs in the wireless telecommunications field.

www.staffing.net
This site allows job seekers to discover openings in the world of wireless technology and communications.

www.wiwc.org
This site’s focus is wireless communication job searching for women.

www.firstsearch.com
At this site a job seeker is able to discover part-time, full-time and salary-based opportunities in the wireless industry.

www.pcsjobs.com
This is the site for The Caradyne Group, which is an executive search firm that focuses on finding job seekers wireless job positions.

www.cnijoblink.com
CNI Career Networks offers confidential, no-charge job placement in the wireless and telecommunications industries.

**SUMMARY**

- The Internet can improve an employer’s ability to recruit employees and help users find career opportunities worldwide.
- Job seekers can learn how to write a resume and cover letter, post them online and search through job listings to find the jobs that best suit their needs.
- Employers can post jobs that can be searched by an enormous pool of applicants.
- Job seekers can store and distribute resumes digitally, receive e-mail notification of possible positions, use salary and relocation calculators, consult job coaches and use self-assessment tools when searching for a job on the Web.
- There are approximately 40,000 career-advancement services on the Internet today.
- Finding a job online can greatly reduce the amount of time spent applying for a position. Potential candidates can also learn more about a company by visiting its Web site.
- Most sites are free to job seekers. These sites typically generate their revenues by charging employers who post their job opportunities, and by selling advertising space on their Web pages.
- Sites such as Vault.com and WetFeet.com allow job seekers to post questions about employers and positions in chat rooms and on bulletin boards.
- On many recruitment sites, the match of a resume to a position is conducted with resume-filtering software.
- A lack of confidentiality is a disadvantage of online career services.
- According to recent studies, the number of individuals researching employment positions through means other than the Internet, such as referrals, newspapers and temporary agencies, far outweighs the number of Internet job seekers.
- Career sites designed for individuals seeking technical positions are among the most popular online career sites.
- Online contracting services allow businesses to post positions for which they wish to hire outside resources, and allow individuals to identify projects that best suit their interests, schedules and skills.
- The Internet provides students and young professionals with some of the necessary tools to get them started in the job market. The target market is made up of individuals still in school and seek-
ing internships, individuals who are just graduating and individuals who have been in the workforce for a few years.

- There are a number of Web sites that offer features that enhance job searches, prepare users to search online, help design applicants’ resumes or help users calculate the cost of relocating.
- Web recruiting reaches a much larger audience than posting an advertisement in the local newspaper.
- There are a variety of sites that allow employers to post jobs online. Some of these sites require a fee, which generally runs between $100–200. Postings remain on the Web site for approximately 30–60 days.
- Employers should try to post to sites that are most likely to be visited by eligible candidates.
- When designing a job posting, defining what makes a job position unique and including information such as benefits and salary might convince a qualified candidate to further investigate the position.
- The Internet hosts demographic-specific sites for employers seeking to increase diversity in the workplace.
- The Internet has provided employers with a cost-effective means of testing their prospective employees in such categories as decision making, problem solving and personality.

**TERMINOLOGY**

corporate culture          open-ended question
demographic               pay-per-hire
end-to-end recruiting solutions
entry-level position      request for proposal (RFP)
online contracting service resume-filtering software

**SELF-REVIEW EXERCISES**

**H.1** State whether each of the following is true or false. If false, explain why.
a) Online contracting services allow businesses to post job listings for specific projects that can be viewed by job seekers over the Web.
b) Employment networks are Web sites designed to provide information on a selected company to better inform job seekers of the corporate environment.
c) The large number of applications received over the Internet is considered an advantage by most online recruiters.
d) There is a greater number of individuals searching for work on the Web than through all other mediums combined.
e) Sixteen percent of America’s workforce is categorized as independent contractors.

**H.2** Fill in the blanks in each of the following statements:
a) There are approximately online career services Web sites on the Internet today.
b) The Internet hosts demographic-specific sites for employers seeking to increase in the workplace.
c) In the 24 hours following the Super Bowl, job searches occurred on Monster.com.
d) Many recruitment sites use to filter through received resumes.
e) Employers should try to post to sites that are most likely to be visited by candidates.
ANSWERS TO SELF-REVIEW EXERCISES

H.1 a) True. b) True. c) False. The large number of applicants reduces the amount of time a recruiter can spend interviewing and making decisions. Despite screening processes, many highly qualified applicants can be overlooked. d) False. The number of individuals researching employment positions through other means, such as referrals, newspapers and temporary agencies, far outweighs the number of Internet job seekers. e) False. Six percent of America’s workforce is categorized as independent consultants.

H.2 a) 40,000. b) diversity. c) 5 million. d) resume-filtering software. e) eligible.

EXERCISES

H.3 State whether each of the following is true or false. If false, explain why.
   a) RFP is the acronym for request for proposal.
   b) The Internet has provided employers with a cost-effective means of testing their prospective employees in such categories as decision making, problem solving and personality.
   c) Online job recruiting can completely replace other means of hiring employees.
   d) Posting a job online is less expensive than placing ads in more traditional media.
   e) A lack of confidentiality is a disadvantage of online career services.

H.4 Fill in the blanks in each of the following statements:
   a) Finding a job online can greatly _________ the amount of time spent applying for a position.
   b) _________ is an example of a Web site in which contractors can bid on projects.
   c) When designing a job posting, defining what makes the position unique and including information such as _________ and _________ might convince a qualified candidate to further investigate the position.
   d) The Internet hosts _________ for employers seeking to increase diversity in the workplace.
   e) The Internet provides employers with a cost-effective means of testing their prospective employees in such categories as _________, _________ and _________.

H.5 Define the following
   a) corporate culture
   b) pay-per-hire
   c) request for proposal (RFP)
   d) resume-filtering software

H.6 (Class discussion). In this appendix, we discuss the shortcomings and advantages of recruiting on the Internet. Using the text, additional reading material and personal accounts answer the following questions. Be prepared to discuss your answers.
   a) Do you think finding a job is easier on the Web? Why or why not?
   b) What disadvantages can you identify?
   c) What are some of the advantages?
   d) Which online recruiting services do you think will be most successful? Why?

H.7 Many of the career services Web sites we have discussed in this appendix offer resume-building capabilities. Begin building your resume, choosing an objective that is of interest to you. Think of your primary concerns. Are you searching for a paid internship or a volunteer opportunity? Do you have a specific location in mind? Do you have an opportunity for future employment? Are stock options important to you? Find several entry-level jobs that meet your requirements. Write a short summary of your results. Include any obstacles and opportunities.
In this appendix, we have discussed online contracting opportunities. Visit FreeAgent (www.freeagent.com) and create your own e.portfolio, or visit eLance (www.elance.com) and search the requests for proposals for contracting opportunities that interest you.

In this appendix, we have discussed many career services Web sites. Choose three sites. Explore the opportunities and resources offered by the sites. Visit any demonstrations, conduct a job search, build your resume and calculate your salary or relocation expenses. Answer the following questions.

a) Which site provides the best service? Why?
b) What did you like? Dislike?
c) Write a brief summary of your findings, including descriptions of any features that you would add.

WORKS CITED
The notation <www.domain-name.com> indicates that the citation is for information found at the Web site.

2. J. Gaskin, 50.
4. <www.vault.com>
5. M. Berger, 224.
8. <www.webhire.com>
11. <www.jobsonline.com>
12. <www.driveway.com>
13. <www.cruelworld.com>
16. <www.experience.com>
Objectives

- To become familiar with Unicode.
- To discuss the mission of the Unicode Consortium.
- To discuss the design basis of Unicode.
- To understand the three Unicode encoding forms: UTF-8, UTF-16 and UTF-32.
- To introduce characters and glyphs.
- To discuss the advantages and disadvantages of using Unicode.
- To provide a brief tour of the Unicode Consortium’s Web site.
I.1 Introduction

The use of inconsistent character encodings (i.e., numeric values associated with characters) when developing global software products causes serious problems because computers process information using numbers. For instance, the character “a” is converted to a numeric value so that a computer can manipulate that piece of data. Many countries and corporations have developed their own encoding systems that are incompatible with the encoding systems of other countries and corporations. For example, the Microsoft Windows operating system assigns the value 0xC0 to the character “A with a grave accent” while the Apple Macintosh operating system assigns that same value to an upside-down question mark. This results in the misrepresentation and possible corruption of data because data is not processed as intended.

In the absence of a widely-implemented universal character encoding standard, global software developers had to localize their products extensively before distribution. Localization includes the language translation and cultural adaptation of content. The process of localization usually includes significant modifications to the source code (such as the conversion of numeric values and the underlying assumptions made by programmers), which results in increased costs and delays releasing the software. For example, some English-speaking programmers might design global software products assuming that a single character can be represented by one byte. However, when those products are localized in Asian markets, the programmer’s assumptions are no longer valid, thus the majority, if not the entirety, of the code needs to be rewritten. Localization is necessary with each release of a version. By the time a software product is localized for a particular market, a newer version, which needs to be localized as well, is ready for distribution. As a result, it is cumbersome and costly to produce and distribute global software products in a market where there is no universal character encoding standard.

In response to this situation, the Unicode Standard, an encoding standard that facilitates the production and distribution of software, was created. The Unicode Standard outlines a specification to produce consistent encoding of the world’s characters and symbols. Software products which handle text encoded in the Unicode Standard need to be localized, but the localization process is simpler and more efficient because the numeric values need not be converted and the assumptions made by programmers about the character encoding are universal. The Unicode Standard is maintained by a non-profit organization called the
Unicode Consortium, whose members include Apple, IBM, Microsoft, Oracle, Sun Microsystems, Sybase and many others.

When the Consortium envisioned and developed the Unicode Standard, they wanted an encoding system that was universal, efficient, uniform and unambiguous. A universal encoding system encompasses all commonly used characters. An efficient encoding system allows text files to be parsed easily. A uniform encoding system assigns fixed values to all characters. An unambiguous encoding system represents a given character in a consistent manner. These four terms are referred to as the Unicode Standard design basis.

I.2 Unicode Transformation Formats

Although Unicode incorporates the limited ASCII character set (i.e., a collection of characters), it encompasses a more comprehensive character set. In ASCII each character is represented by a byte containing 0s and 1s. One byte is capable of storing the binary numbers from 0 to 255. Each character is assigned a number between 0 and 255, thus ASCII-based systems can support only 256 characters, a tiny fraction of world’s characters. Unicode extends the ASCII character set by encoding the vast majority of the world’s characters. The Unicode Standard encodes all of those characters in a uniform numerical space from 0 to 10FFFF hexadecimal. An implementation will express these numbers in one of several transformation formats, choosing the one that best fits the particular application at hand.

Three such formats are in use, called UTF-8, UTF-16 and UTF-32, depending on the size of the units—in bits—being used. UTF-8, a variable width encoding form, requires one to four bytes to express each Unicode character. UTF-8 data consists of 8-bit bytes (sequences of one, two, three or four bytes depending on the character being encoded) and is well suited for ASCII-based systems when there is a predominance of one-byte characters (ASCII represents characters as one-byte). Currently, UTF-8 is widely implemented in UNIX systems and in databases.

The variable width UTF-16 encoding form expresses Unicode characters in units of 16-bits (i.e., as two adjacent bytes, or a short integer in many machines). Most characters of Unicode are expressed in a single 16-bit unit. However, characters with values above FFFF hexadecimal are expressed with an ordered pair of 16-bit units called surrogates. Surrogates are 16-bit integers in the range D800 through DFFF, which are used solely for the purpose of “escaping” into higher numbered characters. Approximately one million characters can be expressed in this manner. Although a surrogate pair requires 32-bits to represent characters, it is space-efficient to use these 16-bit units. Surrogates are rare characters in current implementations. Many string-handling implementations are written in terms of UTF-16. [Note: Details and sample-code for UTF-16 handling are available on the Unicode Consortium Web site at www.unicode.org.]

Implementations that require significant use of rare characters or entire scripts encoded above FFFF hexadecimal, should use UTF-32, a 32-bit fixed-width encoding form that usually requires twice as much memory as UTF-16 encoded characters. The major advantage of the fixed-width UTF-32 encoding form is that it uniformly expresses all characters, so it is easy to handle in arrays.

There are few guidelines that state when to use a particular encoding form. The best encoding form to use depends on computer systems and business protocols, not on the data itself. Typically, the UTF-8 encoding form should be used where computer systems and business protocols require data to be handled in 8-bit units, particularly in legacy systems.
being upgraded because it often simplifies changes to existing programs. For this reason, UTF-8 has become the encoding form of choice on the Internet. Likewise, UTF-16 is the encoding form of choice on Microsoft Windows applications. UTF-32 is likely to become more widely used in the future as more characters are encoded with values above FFFF hexadecimal. Also, UTF-32 requires less sophisticated handling than UTF-16 in the presence of surrogate pairs.

Figure K.1 shows the different ways in which the three encoding forms handle character encoding.

I.3 Characters and Glyphs

The Unicode Standard consists of characters, written components (i.e., alphabets, numbers, punctuation marks, accent marks, etc.) that can be represented by numeric values. Examples of characters include: U+0041 LATIN CAPITAL LETTER A. In the first character representation, U+yyyy is a code value, in which U+ refers to Unicode code values, as opposed to other hexadecimal values. The yyyy represents a four-digit hexadecimal number of an encoded character. Code values are bit combinations that represent encoded characters. Characters are represented using glyphs, various shapes, fonts and sizes for displaying characters. There are no code values for glyphs in the Unicode Standard. Examples of glyphs are shown in Fig. K.2.

The Unicode Standard encompasses the alphabets, ideographs, syllabaries, punctuation marks, diacritics, mathematical operators, etc. that comprise the written languages and scripts of the world. A diacritic is a special mark added to a character to distinguish it from another letter or to indicate an accent (e.g., in Spanish, the tilde “~” above the character “n”). Currently, Unicode provides code values for 94,140 character representations, with more than 880,000 code values reserved for future expansion.

<table>
<thead>
<tr>
<th>Character</th>
<th>UTF-8</th>
<th>UTF-16</th>
<th>UTF-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATIN CAPITAL LETTER A</td>
<td>0x41</td>
<td>0x0041</td>
<td>0x00000041</td>
</tr>
<tr>
<td>GREEK CAPITAL LETTER</td>
<td>0xCD</td>
<td>0x0391</td>
<td>0x00000391</td>
</tr>
<tr>
<td>ALPHA</td>
<td>0x91</td>
<td>0x0391</td>
<td>0x00000391</td>
</tr>
<tr>
<td>CJK UNIFIED IDEOGRAPH-</td>
<td>0xE4</td>
<td>0x4E95</td>
<td>0x00004E95</td>
</tr>
<tr>
<td>4E95</td>
<td>0xBA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OLD ITALIC LETTER A</td>
<td>0xF0</td>
<td>0x080 0x83</td>
<td>0x00010300</td>
</tr>
</tbody>
</table>

Fig. I.1 Correlation between the three encoding forms.

Fig. I.2 Various glyphs of the character A.
I.4 Advantages/Disadvantages of Unicode

The Unicode Standard has several significant advantages that promote its use. One is the impact it has on the performance of the international economy. Unicode standardizes the characters for the world’s writing systems to a uniform model that promotes transferring and sharing data. Programs developed using such a schema maintain their accuracy because each character has a single definition (i.e., a is always U+0061, % is always U+0025). This enables corporations to manage the high demands of international markets by processing different writing systems at the same time. Also, all characters can be managed in an identical manner, thus avoiding any confusion caused by different character code architectures. Moreover, managing data in a consistent manner eliminates data corruption, because data can be sorted, searched and manipulated using a consistent process.

Another advantage of the Unicode Standard is portability (i.e., software that can execute on disparate computers or with disparate operating systems). Most operating systems, databases, programming languages and Web browsers currently support, or are planning to support, Unicode.

A disadvantage of the Unicode Standard is the amount of memory required by UTF-16 and UTF-32. ASCII character sets are 8-bits in length, so they require less storage than the default 16-bit Unicode character set. However, the double-byte character set (DBCS) and the multi-byte character set (MBCS) that encode Asian characters (ideographs) require two to four bytes, respectively. In such instances, the UTF-16 or the UTF-32 encoding forms may be used with little hindrance on memory and performance.

Another disadvantage of Unicode is that although it includes more characters than any other character set in common use, it does not yet encode all of the world’s written characters. Another disadvantage of the Unicode Standard is that UTF-8 and UTF-16 are variable width encoding forms, so characters occupy different amounts of memory.

I.5 Unicode Consortium’s Web Site

If you would like to learn more about the Unicode Standard, visit www.unicode.org. This site provides a wealth of information about the Unicode Standard that is insightful to those new to Unicode. Currently, the home page is organized into various sections—New to Unicode, General Information, The Consortium, The Unicode Standard, Work in Progress and For Members.

The New to Unicode section consists of two subsections: What is Unicode and How to Use this Site. The first subsection provides a technical introduction to Unicode by describing design principles, character interpretations and assignments, text processing and Unicode conformance. This subsection is recommended reading for anyone new to Unicode. Also, this subsection provides a list of related links that provide the reader with additional information about Unicode. The How to Use this Site subsection contains information about using and navigating the site as well hyperlinks to additional resources.

The General Information section contains six subsections: Where is my Character, Display Problems, Useful Resources, Enabled Products, Mail Lists and Conferences. The main areas covered in this section include a link to the Unicode code charts (a complete listing of code values) assembled by the Unicode Consortium as well as a detailed outline on how to locate an encoded character in the code chart. Also, the section contains advice on how to configure different operating systems and Web browsers so that
the Unicode characters can be viewed properly. Moreover, from this section, the user can navigate to other sites that provide information on various topics such as, fonts, linguistics and other standards such as the Armenian Standards Page and the Chinese GB 18030 Encoding Standard.

The Consortium section consists of five subsections: Who we are, Our Members, How to Join, Press Info and Contact Us. This section provides a list of the current Unicode Consortium members as well as information on how to become a member. Privileges for each member type—full, associate, specialist and individual—and the fees assessed to each member are listed here.

The Unicode Standard section consists of nine subsections: Start Here, Latest Version, Technical Reports, Code Charts, Unicode Data, Update & Errata, Unicode Policies, Glossary and Technical FAQ. This section describes the updates applied to the latest version of the Unicode Standard as well as categorizing all defined encoding. The user can learn how the latest version has been modified to encompass more features and capabilities. For instance, one enhancement of Version 3.1 is that it contains additional encoded characters. Also, if users are unfamiliar with vocabulary terms used by the Unicode Consortium, then they can navigate to the Glossary subsection.

The Work in Progress section consists of three subsections: Calendar of Meetings, Proposed Characters and Submitting Proposals. This section presents the user with a catalog of the recent characters included into the Unicode Standard scheme as well as those characters being considered for inclusion. If users determine that a character has been overlooked, then they can submit a written proposal for the inclusion of that character. The Submitting Proposals subsection contains strict guidelines that must be adhered to when submitting written proposals.

The For Members section consists of two subsections: Member Resources and Working Documents. These subsections are password protected; only consortium members can access these links.

### 1.6 Using Unicode

Numerous programming languages (e.g., C, Java, JavaScript, Perl, Visual Basic, etc.) provide some level of support for the Unicode Standard. Figure K.3 shows a Java program that prints the text “Welcome to Unicode!” in eight different languages: English, Russian, French, German, Japanese, Portuguese, Spanish and Traditional Chinese. [Note: The Unicode Consortium’s Web site contains a link to code charts that lists the 16-bit Unicode code values.]

```
1 // Fig. K.3: Unicode.java
2 // Demonstrating how to use Unicode in Java programs.
3
4 // Java core packages
5 import java.awt.*;
6
7 // Java extension packages
8 import javax.swing.*;
9
```

Fig. I.3 Java program that uses Unicode encoding (part 1 of 3).
public class Unicode extends JFrame {
    private JLabel english, chinese, cyrillic, french, german,
    japanese, portuguese, spanish;

    // Unicode constructor
    public Unicode() {
        super( "Demonstrating Unicode" );

        // get content pane and set its layout
        Container container = getContentPane();
        container.setLayout( new GridLayout( 8, 1 ) );

        // JLabel constructor with a string argument
        english = new JLabel( \u0057\u0065\u006C\u006F
                \u006D\u0065\u0020\u0074\u006F\u0020Unicode\u0021 );
        english.setToolTipText( "This is English" );
        container.add( english );

        chinese = new JLabel( \u6B22\u8FCE\u4F7F\u7528
                \u0020Unicode\u0021 );
        chinese.setToolTipText( "This is Traditional Chinese" );
        container.add( chinese );

        cyrillic = new JLabel( \u0414\u043E\u0431\u0440\u043E
                \u0020\u043F\u043E\u0436\u0430\u043B\u043E\u0432
                \u0430\u0422\u044A\u0020\u0432\u0020Unicode\u0021 );
        cyrillic.setToolTipText( "This is Russian" );
        container.add( cyrillic );

        french = new JLabel( \u0042\u0065\u006E\u0076
                \u0065\u006E\u0075\u0065\u0020\u0061\u0075\u0020
                Unicode\u0021 );
        french.setToolTipText( "This is French" );
        container.add( french );

        german = new JLabel( \u0057\u0069\u006C\u006F
                \u006D\u0065\u0020\u007A\u0075\u0020
                Unicode\u0021 );
        german.setToolTipText( "This is German" );
        container.add( german );

        japanese = new JLabel( "\u3078\u3087\u3045\u3053" +
                "\u305D" );
        japanese.setToolTipText( "This is Japanese" );
        container.add( hiragana );

        portuguese = new JLabel( "\u0053\u00E9\u006A\u0020
                \u0042\u0065\u006D\u0076\u006F\u0020Unicode\u0021" );
        portuguese.setToolTipText( "This is Portuguese" );
        container.add( portuguese );
    }
}

Fig. I.3 Java program that uses Unicode encoding (part 2 of 3).
The `Unicode.java` program uses escape sequences to represent characters. An escape sequence is in the form `\uyyyy`, where `yyyy` represents the four-digit hexadecimal code value. Lines 24 and 25 contain the series of escape sequences necessary to print “Welcome to Unicode!” in English. The first escape sequence (`\u0057`) equates to the character “W,” the second escape sequence (`\u0065`) equates to the character “e,” and so on. The `\u0020` escape sequence (line 25) is the encoding for the space character. The `\u0074` and `\u006F` escape sequences equate to the word “to.” Note that “Unicode” is not encoded because it is a registered trademark and has no equivalent translation in most languages. Line 25 also contains the `\u0021` escape sequence for the exclamation mark (!).

Lines 29–65 contain the escape sequences for the other seven languages. The English, French, German, Portuguese and Spanish characters are located in the Basic Latin block, the Japanese characters are located in the Hiragana block, the Russian characters are located in the Cyrillic block and the Traditional Chinese characters are located in the CJK Unified Ideographs block.

[Note: To display the output of `Unicode.java` properly, copy the `font.properties` file to the `font.properties` files (located in the C:\Program Files\JavaSoft\JRE\1.3.1\lib and in the C:\jdk1.3.1\jre\lib directories). Save the contents of `font.properties` prior to overwriting them with the contents from `font.properties.zh`.]
I.7 Character Ranges

The Unicode Standard assigns code values, which range from 0000 (Basic Latin) to E007F (Tags), to the written characters of the world. Currently, there are code values for 94,140 characters. To simplify the search for a character and its associated code value, the Unicode Standard generally groups code values by script and function (i.e., Latin characters are grouped in a block, mathematical operators are grouped in another block, etc.). As a rule, a script is a single writing system that is used for multiple languages (e.g., the Latin script is used for English, French, Spanish, etc.). The Code Charts page on the Unicode Consortium Web site lists all the defined blocks and their respective code values. Figure K.4 lists some blocks (scripts) from the Web site and their range of code values.

**SUMMARY**

- Before Unicode, software developers were plagued by the use of inconsistent character encoding (i.e., numeric values for characters). Most countries and organizations had their own encoding systems, which were incompatible. A good example is the individual encoding systems on the Windows and Macintosh platforms.

<table>
<thead>
<tr>
<th>Script</th>
<th>Range of Code Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arabic</td>
<td>U+0600–U+06FF</td>
</tr>
<tr>
<td>Basic Latin</td>
<td>U+0000–U+007F</td>
</tr>
<tr>
<td>Bengali (India)</td>
<td>U+0980–U+09FF</td>
</tr>
<tr>
<td>Cherokee (Native America)</td>
<td>U+13A0–U+13FF</td>
</tr>
<tr>
<td>CJK Unified Ideographs (East Asia)</td>
<td>U+4E00–U+9FAF</td>
</tr>
<tr>
<td>Cyrillic (Russia and Eastern Europe)</td>
<td>U+0400–U+04FF</td>
</tr>
<tr>
<td>Ethiopic</td>
<td>U+1200–U+137F</td>
</tr>
<tr>
<td>Greek</td>
<td>U+0370–U+03FF</td>
</tr>
<tr>
<td>Hangul Jamo (Korea)</td>
<td>U+1100–U+11FF</td>
</tr>
<tr>
<td>Hebrew</td>
<td>U+0590–U+05FF</td>
</tr>
<tr>
<td>Hiragana (Japan)</td>
<td>U+3040–U+309F</td>
</tr>
<tr>
<td>Khmer (Cambodia)</td>
<td>U+1780–U+17FF</td>
</tr>
<tr>
<td>Lao (Laos)</td>
<td>U+0E80–U+0EFF</td>
</tr>
<tr>
<td>Mongolian</td>
<td>U+1800–U+18AF</td>
</tr>
<tr>
<td>Myanmar</td>
<td>U+1000–U+109F</td>
</tr>
<tr>
<td>Ogham (Ireland)</td>
<td>U+1680–U+169F</td>
</tr>
<tr>
<td>Runic (Germany and Scandinavia)</td>
<td>U+16A0–U+16FF</td>
</tr>
<tr>
<td>Sinhala (Sri Lanka)</td>
<td>U+0D80–U+0DFF</td>
</tr>
<tr>
<td>Telugu (India)</td>
<td>U+0C00–U+0C7F</td>
</tr>
<tr>
<td>Thai</td>
<td>U+0E00–U+0E7F</td>
</tr>
</tbody>
</table>
• Computers process data by converting characters to numeric values. For instance, the character “a” is converted to a numeric value so that a computer can manipulate that piece of data.

• Localization of global software requires significant modifications to the source code, which results in the increased cost and delays releasing the product.

• Localization is necessary with each release of a version. By the time a software product is localized for a particular market, a newer version, which needs to be localized as well, is ready for distribution. As a result, it is cumbersome and costly to produce and distribute global software products in a market where there is no universal character encoding standard.

• The Unicode Consortium developed the Unicode Standard in response to the serious problems created by multiple character encodings and the use of those encodings.

• The Unicode Standard facilitates the production and distribution of localized software. It outlines a specification for the consistent encoding of the world’s characters and symbols.

• Software products which handle text encoded in the Unicode Standard need to be localized, but the localization process is simpler and more efficient because the numeric values need not be converted.

• The Unicode Standard is designed to be universal, efficient, uniform and unambiguous.

• A universal encoding system encompasses all commonly used characters; an efficient encoding system parses text files easily; a uniform encoding system assigns fixed values to all characters; and a unambiguous encoding system represents the same character for any given value.

• Unicode extends the limited ASCII character set to include all the major characters of the world.

• Unicode makes use of three Unicode Transformation Formats (UTF): UTF-8, UTF-16 and UTF-32, each of which may be appropriate for use in different contexts.

• UTF-8 data consists of 8-bit bytes (sequences of one, two, three or four bytes depending on the character being encoded) and is well suited for ASCII-based systems when there is a predominance of one-byte characters (ASCII represents characters as one-byte).

• UTF-8 is a variable width encoding form that is more compact for text involving mostly Latin characters and ASCII punctuation.

• UTF-16 is the default encoding form of the Unicode Standard. It is a variable width encoding form that uses 16-bit code units instead of bytes. Most characters are represented by a single 16-bit unit, but some characters require surrogate pairs.

• Without surrogate pairs, the UTF-16 encoding form can only encompass 65,000 characters, but with the surrogate pairs, this is expanded to include over a million characters.

• UTF-32 is a 32-bit encoding form. The major advantage of the fixed-width encoding form is that it uniformly expresses all characters, so that they are easy to handle in arrays and so forth.

• The Unicode Standard consists of characters. A character is any written component that can be represented by a numeric value.

• Characters are represented using glyphs, which are various shapes, fonts and sizes for displaying characters.

• Code values are bit combinations that represent encoded characters. The Unicode notation for a code value is U+yyyy in which U+ refers to the Unicode code values, as opposed to other hexadecimal values. The yyyy represents a four-digit hexadecimal number.

• Currently, the Unicode Standard provides code values for 94,140 character representations.

• An advantage of the Unicode Standard is its impact on the overall performance of the international economy. Applications that conform to an encoding standard can be processed easily by computers.

• Another advantage of the Unicode Standard is its portability. Applications written in Unicode can be easily transferred to different operating systems, databases, Web browsers, etc. Most companies currently support, or are planning to support Unicode.
• To obtain more information about the Unicode Standard and the Unicode Consortium, visit www.unicode.org. It contains a link to the code charts, which contain the 16-bit code values for the currently encoded characters.

• Numerous programming languages provide some level of support for the Unicode Standard.

• In Java programs, the \u0020 escape sequence represents a character, where yyyy is the four-digit hexadecimal code value. The \u0020 escape sequence is the universal encoding for the space character.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\uyyyy</td>
<td>portability</td>
</tr>
<tr>
<td>ASCII</td>
<td>script</td>
</tr>
<tr>
<td>block</td>
<td>surrogate</td>
</tr>
<tr>
<td>character</td>
<td>symbol</td>
</tr>
<tr>
<td>character set</td>
<td>unambiguous (Unicode design basis)</td>
</tr>
<tr>
<td>code value</td>
<td>Unicode Consortium</td>
</tr>
<tr>
<td>diacritic</td>
<td>Unicode design basis</td>
</tr>
<tr>
<td>double-byte character set (DBCS)</td>
<td>Unicode Standard</td>
</tr>
<tr>
<td>efficient (Unicode design basis)</td>
<td>Unicode Transformation Format (UTF)</td>
</tr>
<tr>
<td>encode</td>
<td>uniform (Unicode design basis)</td>
</tr>
<tr>
<td>escape sequence</td>
<td>universal (Unicode design basis)</td>
</tr>
<tr>
<td>glyph</td>
<td>UTF-8</td>
</tr>
<tr>
<td>hexadecimal notation</td>
<td>UTF-16</td>
</tr>
<tr>
<td>localization</td>
<td>UTF-32</td>
</tr>
<tr>
<td>multi-byte character set (MBCS)</td>
<td>Unicode Standard</td>
</tr>
</tbody>
</table>

**SELF-REVIEW EXERCISES**

1.1  Fill in the blanks in each of the following.
   a) Global software developers had to ________ their products to a specific market before distribution.
   b) The Unicode Standard is an ________ standard that facilitates the uniform production and distribution of software products.
   c) The four design basis that constitute the Unicode Standard are: ________, ________, ________, and ________.
   d) A ________ is the smallest written component the can be represented with a numeric value.
   e) Software that can execute on different operating systems is said to be ________.

1.2  State whether each of the following is true or false. If false, explain why.
   a) The Unicode Standard encompasses all the world's characters.
   b) A Unicode code value is represented as U+yyyy, where yyyy represents a number in binary notation.
   c) A diacritic is a character with a special mark that emphasizes an accent.
   d) Unicode is portable.
   e) When designing Java programs, the escape sequence is denoted by /uyyyy.

**SELF-REVIEW ANSWERS**

1.1  a) localize. b) encoding. c) universal, efficient, uniform, unambiguous. d) character. e) portable.
1.2   a) False. It encompasses the majority of the world’s characters. b) False. The yyyy represents a hexadecimal number. c) False. A diacritic is a special mark added to a character to distinguish it from another letter or to indicate an accent. d) True. e) False. The escape sequence is denoted by \uyyyy.

**EXERCISES**

1.3   Navigate to the Unicode Consortium Web site ([www.unicode.org](http://www.unicode.org)) and write the hexadecimal code values for the following characters. In which block were they located?
   a) Latin letter ‘Z.’
   b) Latin letter ‘n’ with the ‘tilde (~).’
   c) Greek letter ‘delta.’
   d) Mathematical operator ‘less than or equal to.’
   e) Punctuation symbol ‘open quote (“).’

1.4   Describe the Unicode Standard design basis.

1.5   Define the following terms:
   a) code value.
   b) surrogates.
   c) Unicode Standard.

1.6   Define the following terms:
   a) UTF-8.
   b) UTF-16.
   c) UTF-32.

1.7   Describe a scenario where it is optimal to store your data in UTF-16 format.

1.8   Using the Unicode Standard code values, write a Java program that prints your first and last name. The program should print your name in all uppercase letters and in all lowercase letters. If you know other languages, print your first and last name in those languages as well.
Symbols
% SQL wildcard character 455
% escape sequence for % 601
& 1617
&amp; 1617
&apos; 1617
&amp;gt; 1617
&amp;lt; 1617
&amp;quot; 1617
/ 1617
<!-- and --> XHTML comment delimiter 600
<%-- and --> JSP comment delimiter 600
<% and %> scriptlet delimiter 600
<%! and %> declaration delimiter 600
<%= and %> expression delimiter 600
<%= and %> JSP expression delimiter 597
<%@ and %> directive delimiter 634
<%@ and %> directive delimiter 628
\ escapes sequence for % 601
<jsp:forward> 610
<jsp:getProperty> 618, 620
<jsp:include> 610
<jsp:param> 610, 611, 616
<jsp:plugin> 610, 616
<jsp:setProperty> 618, 622
<jsp:useBean> 628
_SQL wildcard character 455, 456
_jspService method 595

Numerics
127.0.0.1 (localhost) 544
128-bit encryption 389
3D computer graphics 160
3D game 180
3D scene 180, 195
3DES 390

A
abandoned shopping cart 995
absolute method of interface ResultSet 475
abstract 43
abstract class 137
abstract class 91
Abstract Account View.java 91
AbstractAccount View.java 94
AbstractAccountManager.java 102
AbstractAccountManager.java 105
Account TextView.java 93
ACL 1164
Action interface 30, 39, 40, 43, 44, 51, 52, 56, 65, 306
Action.ACCELERATOR_KEY constant 45
Action.ACTION_COMMAND _KEY constant 45
Advanced Encryption Standard (AES) 390
Advantage Hiring, Inc. 1746
advertisement 1589
advertising 559
advjhtpl-taglib.tld 642, 643
AffineTransform class 257, 260
AffineTransformOp 151
agent level 15, 1321
agents
definition 1320
smart 1321
static 1320
alarmclock1.idl 1468
alarmclock2.idl 1478
AlarmClockClient.java 1474
AlarmClockImpl.java 1469
Alert class 772
align attribute of <jsp:plugin> action 614
AllPermission 1208
Alpha 194, 195
alpha value 155
ambient color 195
AmbientLight 168, 195
America's Job Bank 1744
American National Standards Institute (ANSI) 390
American Society for Female Entrepreneurs 1746
apersand (&) 1617
AMS 761
ancestor node 116
AND 461, 462, 498
angle bracket 1613
animation 161, 180, 330, 339, 345, 347, 353, 371
animation bean 322
annotation 837
anonymous inner class 861
anonymous port number 796
ANSI 1708
anti-aliasing 221
ANY 1634, 1636
AOL Instant Messenger 16, 1549, 1589
Apache HTTP Server 534, 543
Apache Software Foundation 533, 543
Apache Tomcat 11
Apache Tomcat server 543, 661
Apache’s SOAP implementation API 1596
API (Application Programming Interface) 1653
Appeal Ballot 1704
Appearance 162, 168, 169, 195, 196
appendChild method 1658, 1659
appendChild method of interface Element 672, 673
Apple Computer, Inc. 1764
applet 761
applet’s graphics context 136
application assembler 848
Application Assembly Tool 1192
Application Deployment Tool 662, 694, 696, 865, 1149, 1153, 1160
application descriptor file 761
application implicit object 598
application management software (AMS) 761, 774, 786
application scope 598, 617, 618
application server 14, 848, 991, 992, 1164, 1166
application service provider (ASP) 905, 906, 925
application-specific exception 1075
Applications Objects 1457, 1502
apply-templates element (template) 1682
Aquent.com 1751
Arc2D.CHORD 144
Arc2D.Double 140, 143
Arc2D.OPEN 144
Arc2D.PIE 144
architectural patterns 25
archive attribute of
  <jsp:plugin> action 614
armed node 197
array 23
ArrayList class 674, 675, 1084
article.xml 1654
ASC 456, 457, 458
ascending order (ASC) 457
ASCII 1764
AssetPieChartView.java 97
asterisk (*) 453, 1631, 1632
asymmetric key 390
Asynchronous Method Invocation (AMI) 1518
asynchronous RPC 1596
AT_BEGIN constant 651, 652
AT_END 652
AT_END constant 652
atomic operation 482
AttendeeCounter.java 1265
!ATTLIST 1636, 1637, 1638, 1639, 1641, 1642, 1643, 1644
Attr interface 1658
attribute 139, 447, 1614
attribute declaration 1636
attribute default 1637
attribute element 1683
Attribute interface 234
attribute list 1636
attribute list declaration 1636
attribute-map 1178
attribute of tag library descriptor 646
AttributedString class 234
audio 532
authentication 10, 388, 389, 392, 406, 407, 417, 418, 423, 1572
authentication server script 417
AuthentiDate.com 407
author.xsl 1693
authorISBN table of books database 448, 449
authors table of books database 448
auto commit state 482, 516
autoFlush attribute of page directive 635
ava Authentication and Authorization Services (JAAS) 6
available beans 379
axis 194
B
back-end interface 1214
BackendInterface.java 1216
background property 329
Basic Latin 1769
Basic Object Adapter (BOA) 1456, 1499, 1514, 1516, 1544
BasicStroke 138, 143
batch processing 515
batch update 515
BatchUpdateException class 516
BEA 1163
bean 9, 322
Bean-Managed Persistence 918
bean-managed persistence 905,
  906, 925
bean-managed transaction demarcation 879, 881
Bean-Managed Transactions 898
Bean Markup Language (BML) 1597
Bean Type 865, 918
Bean type 1151
BeanDescriptor class 369
bean-field 1178
BeanInfo interface 364, 369,
  370, 371
bean-managed persistence 12
beanName attribute of
  <jsp:useBean> action 618
Beans Development Kit (BDK) 379
Beans in the Palette Category dialog 323
Beans tab in the Component Palette 323, 328
Behavior 161, 162, 193, 196, 197
behavior scheduler 196
behavioral design patterns 21
Bid4Geeks.com 1750
BilingualJobs.com 1746
bind a remote object to the registry 798
binding 1437, 1438, 1441, 1448,
  1456, 1469, 1477, 1484
bit 389
bit (size of unit) 1764
BitMask 196
Blackvoices.com 1745
block 389
block cipher 390
Bluetooth 1750
blur 136
BlurFilter.java 153
bodycontent element 642,
  646, 652
BodyContent interface 651
BodyTag interface 649
<table>
<thead>
<tr>
<th><strong>Index</strong></th>
<th>page 1777</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BodyTagSupport</strong></td>
<td>class 638, 640, 647</td>
</tr>
<tr>
<td><strong>book.xsl</strong></td>
<td>664, 1692</td>
</tr>
<tr>
<td><strong>BookBean.java</strong></td>
<td>663, 670</td>
</tr>
<tr>
<td><strong>books</strong></td>
<td>database 447, 662</td>
</tr>
<tr>
<td><strong>books</strong></td>
<td>database table relationships 451</td>
</tr>
<tr>
<td><strong>books.jsp</strong></td>
<td>663, 674, 674</td>
</tr>
<tr>
<td><strong>BookServlet.java</strong></td>
<td>663, 674</td>
</tr>
<tr>
<td><strong>bookstore case study</strong></td>
<td>11, 661, 662</td>
</tr>
<tr>
<td><strong>bootstrapping</strong></td>
<td>1448, 1551, 1562</td>
</tr>
<tr>
<td><strong>BorderLayout</strong></td>
<td>class 361</td>
</tr>
<tr>
<td><strong>bottom tier</strong></td>
<td>992, 1366</td>
</tr>
<tr>
<td><strong>bound property</strong></td>
<td>347, 352, 353, 355, 364, 370</td>
</tr>
<tr>
<td><strong>bound-property event</strong></td>
<td>370</td>
</tr>
<tr>
<td><strong>bound-property listener</strong></td>
<td>352</td>
</tr>
<tr>
<td><strong>BoundedShapeController.java</strong></td>
<td>266</td>
</tr>
<tr>
<td><strong>bounding volume</strong></td>
<td>168, 197</td>
</tr>
<tr>
<td><strong>BoundingBox</strong></td>
<td>169</td>
</tr>
<tr>
<td><strong>BoundingSphere</strong></td>
<td>168, 169, 180</td>
</tr>
<tr>
<td><strong>Box</strong></td>
<td>168, 169, 195</td>
</tr>
<tr>
<td><strong>branch graph</strong></td>
<td>162, 167</td>
</tr>
<tr>
<td><strong>BranchGroup</strong></td>
<td>162, 167, 169, 180</td>
</tr>
<tr>
<td><strong>Brassringcampus.com</strong></td>
<td>1753</td>
</tr>
<tr>
<td><strong>brick-and-mortar store</strong></td>
<td>995</td>
</tr>
<tr>
<td><strong>Bridge design pattern</strong></td>
<td>23</td>
</tr>
<tr>
<td><strong>broadcaster MBean</strong></td>
<td>1343, 1356</td>
</tr>
<tr>
<td><strong>BrowserLauncher.java</strong></td>
<td>68</td>
</tr>
<tr>
<td><strong>brute-force cracking</strong></td>
<td>394</td>
</tr>
<tr>
<td><strong>buffer</strong></td>
<td>attribute of page directive 635</td>
</tr>
<tr>
<td><strong>BufferedImage</strong></td>
<td>138, 143, 147, 149, 150, 152, 153, 154, 155, 159, 218, 240, 1299</td>
</tr>
<tr>
<td><strong>BufferedImageOp</strong></td>
<td>139, 150, 151</td>
</tr>
<tr>
<td><strong>BufferedReader</strong></td>
<td>class 724, 756, 797</td>
</tr>
<tr>
<td><strong>Bug2Bug.com</strong></td>
<td>shopping cart 662</td>
</tr>
<tr>
<td><strong>builder tool</strong></td>
<td>9, 322, 323, 330, 339, 346, 353, 364, 369, 370, 371, 374, 375</td>
</tr>
<tr>
<td><strong>built-in entity</strong></td>
<td>1617</td>
</tr>
<tr>
<td><strong>business logic</strong></td>
<td>13, 577, 991, 992, 993, 995, 1010</td>
</tr>
<tr>
<td><strong>business logic method</strong></td>
<td>1075, 1119</td>
</tr>
<tr>
<td><strong>business method</strong></td>
<td>847, 1095</td>
</tr>
<tr>
<td><strong>business rule</strong></td>
<td>993</td>
</tr>
<tr>
<td><strong>business-to-business (B2B) transaction</strong></td>
<td>1628</td>
</tr>
<tr>
<td><strong>ButtonGroup</strong></td>
<td>class 944</td>
</tr>
<tr>
<td><strong>ButtonModel</strong></td>
<td>interface 87</td>
</tr>
<tr>
<td><strong>bytecode verifier</strong></td>
<td>410, 430</td>
</tr>
<tr>
<td><strong>ByteLookUpTable</strong></td>
<td>151</td>
</tr>
<tr>
<td><strong>BytesMessage</strong></td>
<td>class 939</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>1616, 1706, 1707</td>
</tr>
<tr>
<td><strong>C++</strong></td>
<td>17, 1618, 1706, 1707</td>
</tr>
<tr>
<td><strong>C++ libraries</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>552</td>
</tr>
<tr>
<td><strong>CachedRowSet</strong></td>
<td>class 520</td>
</tr>
<tr>
<td><strong>cache-strategy</strong></td>
<td>1168</td>
</tr>
<tr>
<td><strong>Caesar Cipher</strong></td>
<td>388</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td>class 1601</td>
</tr>
<tr>
<td><strong>CallableStatement</strong></td>
<td>interface 515, 517</td>
</tr>
<tr>
<td><strong>CallDoubleMethod</strong></td>
<td>function of C table JNIEnv 1713</td>
</tr>
<tr>
<td><strong>CallObjectMethod</strong></td>
<td>function of C table JNIEnv 1713</td>
</tr>
<tr>
<td><strong>CallStaticMethod</strong></td>
<td>function of C table JNIEnv 1716</td>
</tr>
<tr>
<td><strong>CallMethod</strong></td>
<td>function of C table JNIEnv 1713</td>
</tr>
<tr>
<td><strong>CampusCareerCenter.com</strong></td>
<td>1753</td>
</tr>
<tr>
<td><strong>Candidate.java</strong></td>
<td>969</td>
</tr>
<tr>
<td><strong>CandidateEJB.java</strong></td>
<td>970</td>
</tr>
<tr>
<td><strong>CandidateHome.java</strong></td>
<td>969</td>
</tr>
<tr>
<td><strong>Canvas</strong></td>
<td>class 772</td>
</tr>
<tr>
<td><strong>Canvas3D</strong></td>
<td>163, 167, 180</td>
</tr>
<tr>
<td><strong>capability bit</strong></td>
<td>168, 180</td>
</tr>
<tr>
<td><strong>Career.com</strong></td>
<td>1744</td>
</tr>
<tr>
<td><strong>CareerLeader.com</strong></td>
<td>1748</td>
</tr>
<tr>
<td><strong>CareerPath.com</strong></td>
<td>1744</td>
</tr>
<tr>
<td><strong>CareerWeb</strong></td>
<td>1744</td>
</tr>
<tr>
<td><strong>caretaker object</strong></td>
<td>24</td>
</tr>
<tr>
<td><strong>CartItemBean.java</strong></td>
<td>664, 683</td>
</tr>
<tr>
<td><strong>cascaded discovery</strong></td>
<td>1590</td>
</tr>
<tr>
<td><strong>Cascading discovery</strong></td>
<td>1590</td>
</tr>
<tr>
<td><strong>Cascading Style Sheet (CSS)</strong></td>
<td>663</td>
</tr>
<tr>
<td><strong>case study</strong></td>
<td>13, 991</td>
</tr>
<tr>
<td><strong>CDATA</strong></td>
<td>attribute 1637, 1638</td>
</tr>
<tr>
<td><strong>CDATASection</strong></td>
<td>interface 1658</td>
</tr>
<tr>
<td><strong>C++ libraries</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>C++ type</strong></td>
<td>1708, 1709</td>
</tr>
<tr>
<td><strong>character data</strong></td>
<td>1616</td>
</tr>
<tr>
<td><strong>character set</strong></td>
<td>62, 1764</td>
</tr>
<tr>
<td><strong>CharacterData</strong></td>
<td>interface 1658</td>
</tr>
<tr>
<td><strong>Chat.idl</strong></td>
<td>1482, 1496</td>
</tr>
<tr>
<td><strong>ChatClient.java</strong></td>
<td>820</td>
</tr>
<tr>
<td><strong>ChatMessage.java</strong></td>
<td>821</td>
</tr>
<tr>
<td><strong>ChatMessageImpl.java</strong></td>
<td>1497</td>
</tr>
<tr>
<td><strong>ChatServer.java</strong></td>
<td>810</td>
</tr>
<tr>
<td><strong>ChatServerAdministrator.java</strong></td>
<td>816, 1535</td>
</tr>
<tr>
<td><strong>ChatServerImpl.java</strong></td>
<td>811, 1484, 1532</td>
</tr>
<tr>
<td><strong>checked exception</strong></td>
<td>793</td>
</tr>
<tr>
<td><strong>checked RemoteException</strong></td>
<td>797</td>
</tr>
<tr>
<td><strong>checkout</strong></td>
<td>method of interface ShoppingCart 1040</td>
</tr>
<tr>
<td><strong>CheckOutServlet.java</strong></td>
<td>1040</td>
</tr>
<tr>
<td><strong>ChiefMonster</strong></td>
<td>1752</td>
</tr>
<tr>
<td><strong>child element</strong></td>
<td>1614</td>
</tr>
<tr>
<td><strong>child node</strong></td>
<td>116</td>
</tr>
<tr>
<td><strong>children</strong></td>
<td>1614</td>
</tr>
<tr>
<td><strong>ChoiceGroup</strong></td>
<td>776, 778</td>
</tr>
<tr>
<td><strong>choose</strong></td>
<td>element 1688</td>
</tr>
<tr>
<td><strong>cHTML (compact-HTML)</strong></td>
<td>719, 720, 724, 751, 753, 758, 785, 993</td>
</tr>
<tr>
<td><strong>CIM (Common Information Model)</strong></td>
<td>1366</td>
</tr>
<tr>
<td><strong>Cipher</strong></td>
<td>class 404</td>
</tr>
<tr>
<td><strong>CipherInputStream</strong></td>
<td>class 404</td>
</tr>
<tr>
<td><strong>CipherOutputStream</strong></td>
<td>class 405</td>
</tr>
<tr>
<td><strong>CipherOutputStream</strong></td>
<td>class 404</td>
</tr>
<tr>
<td><strong>cipher text</strong></td>
<td>388, 393</td>
</tr>
<tr>
<td><strong>CJK Unified Ideographs</strong></td>
<td>1769</td>
</tr>
<tr>
<td><strong>class</strong></td>
<td>attribute of <a href="">jsp:useBean</a> action 618</td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>class 265, 474</td>
</tr>
<tr>
<td><strong>class loader</strong></td>
<td>410, 430</td>
</tr>
<tr>
<td><strong>class path</strong></td>
<td>342</td>
</tr>
<tr>
<td><strong>class server</strong></td>
<td>1366, 1368</td>
</tr>
</tbody>
</table>
ClassNotFoundException class 466
CLASSPATH (for Jini) 1199
CLDC 758
clearBatch method of interface Statement 516
class 466
CLASSPATH (for Jini) 1199
CLDC 758
clearBatch method of interface Statement 516
client 717, 1549
client connect to server 10, 533
client object 1370
client/server application 533
client-server networking 594
client tier 26, 662, 719, 785, 993, 1366
client/server 532
client/server network 1549, 1550, 1551, 1552, 1560
client/server relationship 532
ClientGUI.java 827
ClientModel.java 1021
ClientNotificationHandler 1343
ClientPrinterManagement.java 1347
clients.dtd 1021
clients.xml 1020
ClientUserAgentHeaders.java 723
clipping 139
clock.jsp 596
clock2.jsp 608
ClockClientGUI.java 1472
close method of interface Cloneable 1382
close method of interface Collection 465
close method of interface Statement 465
Cloudscape 10, 446, 463, 578, 623, 694, 695, 1192
Cloudscape database 978
Cloudscape DBMS 898
CloudscapeDataAccess.java 488
CNN.com 560
code attribute of <jsp:plugin> action 613
code value 1765, 1769, 1771
config 410, 412, 422
codebase attribute of <jsp:plugin> action 614
Codebase field 1201
Coded Name 898, 1153
Collection 1040
Collection interface 244, 1088, 1101
Collections class 244
Collegegrads.com 1753
Collide.java 198
collision 406
collision detection 161, 179, 180, 196, 197
Color 169
color band 154
Color class 96, 361
color manipulation 136
color scale 147, 155
Color3D 169
Color3f 169, 196
ColorConvertOp 151
ColorFilter.java 154
coloring attributes 162
ColorModel 138, 147, 155
ColorSliderPanel.java 357
column 446, 447, 448
column number 454
column number in a result set 454
COM.cloudscape.core.LocalConnectionPoolDataSource 1192
COM.cloudscape.core.RmiJdbcDriver 465, 466, 467, 581
com.sun.jdmk.commpackage 1343
comma (,) 1631, 1636
Command class 781
Command design pattern 40, 220, 776
command line tool 1392
Command object 773, 774, 780, 786
Command.BACK constant 774, 776
Command.OK constant 774, 776
CommandListener interface 774, 778
comment 600
Comment interface 1658
commit 482, 498, 516
commit a response 541
commit method of interface Connection 482, 516
Common Gateway Interface (CGI) 10
Common Information Model (CIM) 1366
Common Object Request Broker Architecture (CORBA) 5, 15, 532, 792, 820, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1447, 1448, 1449, 1451, 1452, 1453, 1454, 1455, 1456, 1458, 1459, 1462, 1466, 1467, 1474, 1476, 1477, 1480, 1481, 1484, 1489, 1492, 1493, 1495, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1509, 1510, 1511, 1513, 1514, 1516, 1523, 1524, 1525, 1527, 1528, 1529, 1530, 1531, 1534, 1535, 1543, 1544
communication framework 1596
Community Review 1703
compatible 155
complex curve 144
component 326, 329, 330, 331
component assembler 9, 322, 323, 330
Component class 51, 57
component environment entries 669
Component Inspector 326
Component Inspector 326, 329, 331, 346, 364
Component Interface Definition Language (CIDL) 1523, 1525
Component Palette 323, 328, 331, 333, 344, 346, 353, 362, 369
ComponentListener interface 257
compositing 139
compositing rule 139
comprehensive job site 1739
compression filter 146
computer security 387
CONCUR_READ_ONLY static constant 475
CONCUR_UPDATABLE static constant 475, 518
concurrency 473
concurrency design patterns 25
concurrency problem 25
conditional section 1644
conditional.dtd 1644
conditional.xml 1645
conditional.xsl 1688
Con 195
config implicit object 598
configure lookup service 1201
  rmid 1201
  Web server 1200
configure a data source 694
connect to a database 464, 470
connect with events 330
Connected Limited Device
  Configuration (CLDC) 758, 781, 786
ConnectException class 807
connecting the dots 323
connection between Java program and database 467
Connection interface 465, 467, 469, 473, 475, 482, 498, 518, 581, 667, 886, 908, 918
Connection Mode 333, 353, 362
connection pool 519
Connection Wizard dialog 333, 353, 362, 364
Connection Wizard idialog 353
ConnectionInfo.lastAutoincrementValue static method 498
ConnectionPoolDataSource interface 519
connector MBeans 1338
consumer 1521
container 1168
Container class 325
container-managed field 920, 1134, 1152
container-managed persistence (CMP) 905, 906, 920, 924, 925, 1085, 1152, 1175
Container-Managed Persistence radio button 1152
container-managed transaction demarcation 879, 886
Container-Managed Transactions 866
container provider 848
Container-Managed 982
container-managed persistence 12
content branch graph 162
content specification 1630
contentType attribute of page directive 636
context root 545, 547, 557
controller 8, 86, 87, 102, 112, 995
controller (in MVC architecture) 25
Controller logic 13
controller logic 991, 1010
creator server 1368, 1369
ControlPanel.java 175
ControlPanel1.java 210
ConvolveOp 151, 152, 153
Cookie class 538, 564
cookie deleted when it expires 561
cookie domain 568
cookie expiration 561
cookie name 564
cookie protocol 568
cookie value 564
cookies disabled 561
CookieSelectLanguage.html 565
CookieServlet.java 561
Cooljobs.com 1753
cordinate system 136, 161, 212
copyright laws 1551
CORBA 820, 860
cordinate system 136, 161, 212
copyright laws 1551
CORBA 820, 860
cordinate system 136, 161, 212
copyright laws 1551
CORBA Component Model (CCM) 1516, 1523, 1524, 1525, 1526, 1543, 1544
CORBA Domains 1457, 1502
CORBA Packages 532
CORBAfacilities 1457, 1502, 1505
CORBAMessageManager.java 1489
CORBAComponentModel 1456, 1457, 1477, 1499, 1500, 1502, 1509, 1516, 1517, 1518, 1525, 1527, 1529, 1543, 1544
corporate culture 1742, 1745
create a Web component 698
create method 847
create method of an EJB home interface 851
create method of class TransactionFactory 1289
create methods of Entity EJB 906
Create new EJB in Application 978
createAttribute method 1658
createCDATASection method 1658
createComment method 1658
createCompatibleWritableRaster method 155
createDefaultDragGestureRecognizer method of class DragSource 302
createElement method 1658
createElement method of interface Document 672, 673
CreateException class 907, 916, 925, 1023
createProcessingInstruction method 1658
createRegistry method of class LocateRegistry 815, 1575
createStatement method of interface Connection 465, 468, 473
createStatement method of interface ResultSet 518
createTextNode method 1658
createTextNode method of interface Document 672, 673
creational design patterns 20
crimson.jar 679
Cruel World 1748
cryptanalysis 393
cryptographic cipher 388
cryptographic standard 390
cryptography 388, 390
cryptologist 393
cryptosystem 388
CSS attribute 666
CSS file 694
CSS specification 666
Ctrl key 1150
curve 144
custom behavior 193
custom event 355
custom tag 638
custom tag attribute 643
custom tag handler 648
custom tag libraries 11
custom tag library 594, 638
custom tag with attributes 645
customer.java 1119
customerEJB.java 1120
customerHome.java 1126
customerModel.java 1127
customize in a builder tool 370, 371
Customizer Dialog 378
Customizer interface 375
customTagAttribute.jsp 644
customTagBody.jsp 647
customTagWelcome.jsp 639
cyclic 142
Cylinder 195

daemon process 1447, 1454
daemon thread 1577
data attribute 447
data component 86
Data Encryption Standard (DES) 390
data pool 1166, 1178
data source 1166, 1168
Data Sources 1192
data structure 23
data tier 992
DataAccessException.java 487
database 445, 452, 534, 577, 756, 1119
database access 533, 577
database driver 467
Database JNDI Name 981, 1155
database management system
(DBMS) 445
database manipulation application 528
Database Name 1192
database resource 695
Database Settings 981
database table 446
database tier 662
DatabaseException.java 1478
DataBuffer 147
DataFlavor class 61, 283, 284, 287
DataFlavor.javaFileListFlavor constant 61
DatagramPacket class 1576, 1577
DataInputStream class 775
DataOutputStream class 779
DataSource class 918
DataSource interface 519, 667, 669
Date class 62, 1391
DateFormat class 62, 606
DBCS (double byte character set) 1766
dbms-column 1178
DB2 446
deadlock 25
Debug class 1409
ddebugException method of class Debug 1409
decipher 389
declaration 596, 600
Decorator design pattern 405, 406
decryption 388, 389, 390, 394
decryption key 391, 392
DEFAULT 606
default event 371
default property 370
default serialization mechanism 793
default servlet 547
default.properties 695
default.properties file 906
DefaultListModel class 107, 1570
DefaultMutableTreeNode class 117, 122, 123
defaultnamespace.xml 1622
DefaultTableModel class 112, 115
DefaultTreeCellEditor class 117
DefaultTreeCellRenderer class 117, 129
DefaultTreeModel class 117, 122
#define 1707, 1708
Deitel Bookstore 13
Deitel Bookstore application architecture 993
Deitel Messenger 809, 820
DeitelDrawing.java 309
DeitelMessenger.java 836, 1493, 1541
delegate 8, 87
delegate-model architecture 86, 87, 107, 111, 115, 803, 8
DELETE FROM 453, 462
delete request 537
DELETE SQL statement 462, 498, 906, 923
deleterow method of interface ResultSet 518
demographic 13, 1118
Dependent Classpath 1193
deploy a Web application 545
deploy an application 694, 709
Deploy Application 868, 1160
deploy session EJB 862
Deployed module location 1193
deployer 848
deploying a service 1597
deploying an EJB 1149
deployment 1393, 1420
deployment descriptor 545, 546, 581, 1168, 1178
Deployment Settings 981
Deployment Settings... button in the Entity tab 1155
DeploymentException class 1395
design pattern 220, 265, 364, 405, 772, 776, 1323, 1381
design patterns xxiv, 9, 20, 22, 24, 25
Design Patterns, Elements of Reusable Object-Oriented Software 20
Destination Type 984
destroy method of interface Servlet 535, 582
destroyApp method of class MIDlet 761
developer.java.sun.com/developer/Books/JDBCTutorial/chapter5.html 521
developer.java.sun.com/developer/earlyAccess/crs 521
developer.java.sun.com/developer/index.html 521
developer.java.sun.com/developer/techDocs/hi/repository 510
diacritic 1765
Dice.com 1749
Diffie, Whitfield 390
diffuse color 195
digital authentication standard 407
digital certificate 407, 409, 423
digital envelope 394
digital signature 10, 406, 407, 413, 415, 1572
Digital Signature Algorithm (DSA) 407
dimension 195
Dimension class 96
Direct3D 160, 161
DirectionalLight 168, 169, 175, 195
directive 595, 634
discarded method of class LookupDiscovery 1209
disconnected RowSet 520
DisconnectListener.java 827
discovered method of class LookupDiscovery 1209
discovered method of interface DiscoveryListener 1399
discovery 1204
multicast discovery 1204, 1209
unicast discovery 1204
discovery process 1397
discovery through invitation 1590
discovery utilities 1232
LookupDiscoveryManager class 1232
LookupLocatorDiscovery class 1232
DiscoveryEvent class 1209, 1235, 1249
DiscoveryGroupManagement.ALL_GROUPS constant 1237
DiscoveryListener interface 1209, 1220, 1232, 1237
DiscoveryManagement interface 1249
Display class 773, 786
Display Console 1368
display-name element 546, 547
Displayable class 772
DisplayAuthors.java 464
DisplayQueryResults.java 476
distributed Web application 11, 661, 662
diversity 1745
file extension 1709
DndConstants.ACTION_COPY constant 57, 62
DndConstants.ACTION_LINK constant 57
DndConstants.ACTION_MOVE constant 57, 62
dDWebBrowser.java 57
doAfterBody method of interface BodyTag 649, 651
!DOCTYPE 1629, 1635, 1636, 1637, 1639, 1641, 1642, 1645
Document 739, 740, 741, 744, 745
Document interface 252, 672, 673, 677, 680, 740, 1019, 1658
Document Object Model (DOM) 680, 1616, 1653
Document Type Declaration 1629
Document Type Definition (DTD) 1628
DocumentBuilderFactory class 677, 680, 741, 1019, 1678, 1679
documentroot 1700
doDelete method of class HttpServlet 537
doEndTag method of interface Tag 641, 651
doGet method 676
doGet method of class HttpServlet 536, 537, 538, 539, 541, 553, 720, 739
dogfriendly.com 1753
doInitBody method of interface BodyTag 648
DOM (Document Object Model) 1616, 1653
DOM API (Application Programming Interface) 1653
DOM-based parser 1653
domain name 1331, 1367
DOMSource class 680, 741, 1019, 1678, 1679
doOptions method of class HttpServlet 537
doPost method of class HttpServlet 536, 537, 538, 553, 564
doPut method of class HttpServlet 537
doStartTag method of interface Tag 641, 642, 651
doTrace method of class HttpServlet 537
drag and drop 56
drag-and-drop 221
drag-and-drop data transfer 56
drag-and-drop operation 56, 57
drag gesture 56
drag source 56
Drag-and-Drop API 9
DragAndDropController.java 276
dragDropActionChanged method of interface DragSourceListener 284
dragDropActionChanged method of interface DropTargetListener 285
dragDropEnd method of interface DragSourceListener 284
dragEnter method of interface DragSourceListener 284
dragEnter method of interface DropTargetListener 61, 285
dragExit method of interface DragSourceListener 284
dragExit method of interface DropTargetListener 62, 285
DragGestureEvent class 283
DragGestureListener interface 276
dragGestureRecognized
method of interface Drag-
GestureListener 283
DragGestureRecognizer
class 276
dragOver method of interface
DragSourceListener
284
DragOver method of interface
DropTargetListener
62, 285
DragSource class 276, 302
DragSourceDragEvent
class 284
DragSourceListener
interface 276
draw 139, 218
drawing application 9
drawing application 9
drawString 139
drawString
139
drivers 1763
driverManager class 465, 467
driveway.com class 1748
drop method of interface
DropTargetListener
57, 283, 285
drop target 56
dropActionChanged method
of interface DropTar-
getListener 62
dropComplete method of class
DropTargetDropEvent
61, 284
DropTarget
class 57, 61, 62, 276
dropTargetDragEvent
class 61, 62, 285
DropTargetDragEvent
class 57, 61
DropTargetDropEvent
interface 283
DropTargetDropEvent
class 285
DropTargetListener
interface 57, 62, 276, 283
DTD (Document Type Definition)
745, 1628, 1629
.dtd extension 1629
DuplicateKeyException
class 925
durable subscription 951
dynamic class downloading 836,
837
dynamic content 594, 599
dynamic HTML 577
Dynamic Invocation Interface
(DII) 1456, 1502, 1510,
1511, 1514
Dynamic MBeans 1323
dynamic service 1369
Dynamic Skeleton Interface (DSI)
1510
dynamic XHTML document 662
dynamicServiceFinder
java 1397
E
EAR file 697
EBNF (Extended Backus-Naur
Form) grammar 1628
e-business application 13, 991
e-commerce model 1022
edge hint 152
Edit menu 39
efficient (Unicode design basis)
1764, 1771
EJB 6, 781, 847
EJB container 847, 848, 849, 939,
1088, 1126, 1145
EJB create method 1155
EJB Display Name 978
EJB finder method 1155
EJB implementation 847
EJB JAR file 1149, 1150
EJB Method 1156
ejb-name 1168
ejb-ref-name 1176, 1190
ejb-reference-description
1190
EJB References 1192
EJB server provider 848
EJB wizard 1152
ejbActivate method of
interface SessionBean,
EntityBean 854
EJBContext interface 853
ejbCreate method 853, 1125
EJBException class 886, 917,
918
ejbFindByPrimaryKey
method 917
EJBHome interface 848
EJBObject interface 848, 906
ejbPassivate method of
interface SessionBean
853
ejbStore 1168
eLance.com 1751
!ELEMENT 1630, 1634, 1635,
1636, 1637, 1639, 1641,
1642, 1644
Element 740, 741, 745, 748, 753
element content 1616, 1630
element element 1683
element group 1632
Element interface 222, 672, 673,
677, 680, 1019, 1658
element type declaration 1630
elements.xsl 1681
Ellipse2D.Double 140, 143,
218
Ellipse2D.Float 140
Ellipse2D.Float class 233
embed HTML element 604, 613
embedded Cloudscape
framework 463
Employee.java 906
EmployeeEJB.java 909, 921
EmployeeEJBclient.java
926
EmployeeHome.java 908
EMPTY 1634, 1644
encipher 389
encoding 1763
encoding declaration 1614
encryption 10, 389, 390, 392
encryption algorithm 393
encryption key 389, 392, 393
der end caps 143
enhancement filter 146
Enterprise Application Archive
(EAR) file 696
Enterprise Applications 1193
Enterprise Bean Class 865,
978, 982
Enterprise Bean Display
Name 865
Enterprise Bean Name 978,
982
enterprise bean provider 848
Enterprise Bean References
984
Enterprise JavaBean 6
Enterprise JavaBeans 12, 13, 847,
991, 993, 1022
ENTITIES 1643
ENTITY 1638, 1642, 1644
Entity 1151, 1153, 1155
definition 1639
definition attribute 1640
defined declaration 1639
entity reference 1617
Entity Settings dialog 918, 978
ENTITY tokenized attribute type 1640
EntityBean interface 908
EntityContext class 917
entityExample.xml 1642
Entry
Name 1261
Entry interface 1220, 1260, 1265, 1266, 1269
entry object 1259
entry point 1379, 1395, 1410, 1428
entry point proxy 1393
Entry utilities 1242
Address 1242
Comment 1242
Location 1242
Name 1242
ServiceInfo 1242
ServiceType 1242
EntryListener.java 1276
enumerated attribute 1638, 1643
Enumeration interface 1034
envelope 1596
environment variable 543, 1392
equals method of class Object 1111
error method
(ErrorHandler) 1664
error page attribute of page directive 622
errorPage attribute of page directive 628, 636
Errors panel in Igniter GUI 1368
escape character 461
escape sequence 600, 601, 1769
EVAL_BODY_INCLUDE constant 642
EVAL_BODY_TAG 651
event 1381
Event class 1382
event driven process 137
event-handling 88
event notification 1343
event publisher 1381
event service 1368, 1369, 1378, 1381, 1399, 1409, 1428, 1521
event set 371
event set name 370
event source 370
event subscriber 1381
event topic 1381, 1382, 1383
EventListener class 355
EventNotHandledException class 1381
EventObject class 355
EventSetDescriptor class 370, 371
eWork® Exchange 1751
Examples
AbstractAccountView.java 91
AbstractDrawingAction.java 306
Account.java 89
AccountBarGraphView.java 94
AccountController.java 102
AccountManager.java 105
AccountTextView.java 93
ActionSample.java 40
ActionSample2.java 72
Address.java 1134
AddressBook.java 503
AddressBookDataAccess.java 486
AddressBookEntry.java 483
AddressBookEntryFrame.java 500
AddressHome.java 1138
AddressModel.java 1139
AddToCartServlet.java 664, 685, 1024
adrotator.java 620
adrotator.jsp 620
advjhtml-taglib.tld 643
alarmclock1.idl 1468
alarmclock2.idl 1478
AlarmClockClient.java 1474
AlarmClockImpl.java 1469
article.xml 1654
AssetPieChartView.java 97
AttendeeCounter.java 1265
author.xsl 1693
BackendInterface.java 1216
banner.html 606
BlurFilter.java 153
book.xml 664, 1692
BookBean.java 663, 670
books.jsp 663, 674
BookServlet.java 664, 676
BoundedShapeController.java 266
BrowserLauncher.java 68
Candidate.java 969
CandidateEJB.java 970
CandidateHome.java 969
CartItemBean.java 664, 683
Changing a pipe character to a comma in a DTD 1636
chapters.xsl 1693
Chat.idl 1482, 1496
ChatClient.java 820
ChatMessage.java 821
ChatMessageImpl.java 1497
ChatServer.java 810
ChatServerAdministrator.java 816, 1535
ChatServerImpl.java 811, 1484, 1532
CheckOutServlet.java 1040
ClientGUI.java 827
ClientModel.java 1021
ClientPrinterManagement.java 1347
clients.dtd 1021
clients.xml 1020
ClientUserAgentHeaders.java 723
clock.jsp 596
clock2.jsp 608
ClockClientGUI.java 1472
CloudscapeDataAccess.java 488
Collide.java 198
ColorFilter.java 154
ColorSliderPanel.java 357
Conditional sections in a DTD 1644
conditional.dtd 1644
conditional.xml 1645
conditional.xsl 1688
ControlPanel.java 175
Index

ControlPanel1.java 210
CookieSelectLanguage.html 565
CookieServlet.java 561
CORBAMessageManager.java 1489
Customer.java 1119
CustomerEJB.java 1120
CustomerHome.java 1126
CustomerModel.java 1127
customTagAttribute.jsp 644
customTagBody.jsp 647
customTagWelcome.jsp 639
DataAccessException.java 487
DatabaseException.java 1478
Declaring attributes 1637
DeitelDrawing.java 309
DeitelMessenger.java 836, 1493, 1541
DisconnectListener.java 827
DisplayAuthors.java 464
DisplayQueryResults.java 476
DnDWebBrowser.java 57
DragAndDropController.java 276
DrawingFileFilter.java 304
DrawingFileReaderWriter.java 245
DrawingInternalFrame.java 287
DrawingModel.java 243
DrawingView.java 254
DynamicServiceFinder.java 1397
elements.xsl 1681
Employee.java 906
EmployeeEJB.java 909, 921
EmployeeEJBClient.java 926
EmployeeHome.java 908
entityExample.xml 1642
EntryListener.java 1276
Example of a mixed-content element 1635
FavoritesWebBrowser.java 45
FileSystemModel.java 123
FileTreeFrame.java 129
Filters.java 1299
forward1.jsp 611
forward2.jsp 612
games.xml 1681
GeneralDiscoveryUtility.java 1237
GetAllProductsServlet.java 1043
GetMessage.java 1599
GetPasswordHintServlet.java 1068
GetProductsServlet.java 1046
GoalDetector.java 205
GradientIcon.java 307
GuestBean.java 623
guestBookErrorPage.jsp 631
guestBookLogin.jsp 626
guestBookTag.jsp 649
guestBookTagExtraInfo.java 652
guestBookView.jsp 629
GuestDataBean.java 624
IDExample.xml 1639
ImageDisplayLayer 1310
ImageEntry.java 1295
ImagePanel.java 147
ImageParser.java 1309
ImageProcessor.java 1295
ImageprocessorMain.java 1301
ImageSeparator.java 1301
ImageSizeException.java 1722
IMConstants.java 1575
IMODETipAnswer.xsl 754
IMODETipQuestion.xsl 751
IMPeer.java 1553
IMPeerImpl.java 1560
IMPeerListener.java 1557
IMSClient.java 1553
IMServiceImpl.java 1555
IMServiceManager.java 1562
include.jsp 608
includeDirective.jsp 637
index.html 663, 665
InterestCalculator.java 850
InterestCalculatorClient.java 854
InterestCalculatorEJB.java 852
InterestCalculatorHome.java 851
intro.xml 1644, 1680
intro.xsl 1679
Invalid XML document 1631
invalid-entityExample.xml 1642
invalid-IDEExample.xml 1641
invalid-mixed.xml 1636
InvertFilter.java 152
Java2DExample.java 156
Java2DImageFilter.java 150
Java3DExample.java 179
Java3DWorld.java 163
Java3DWorld1.java 181
JavaSpaceFinder.java 1262
JNIArrayMain.java 1721
JNIArrayWrapper.h 1718
JNIArrayWrapper.java 1718
JNIArrayWrapperImpl.cpp 1719
JNIImageFrame.java 1731
JNIMethodImpl.java 1710
JNIPanel.java 1727
JNIPIMain.java 1714
JNIPIMethodImplWrapper.h 1711
SeminarioJoinService.java 1249
SeminarioLeaseService.java 1244
SeminarioService.java 1220
SeminarioInterface.java 1215
SeminarioProvider.java 1243
SeminarioProxy.java 1217
SequenceFactory.java 1145
SequenceFactoryEJB.java 1145
SequenceFactoryHome.java 1147
SessionSelectLanguage.html 574
SessionServlet.java 569
Shapes.java 140
Shapes2.java 144
ShapesApplet.java 614
SharpenFilter.java 153
ShoppingCart.java 1076
ShoppingCartEJB.java 1077
ShoppingCartHome.java 1084
Simple XML document containing a message 1613
simpleContact.xml 1671
SimpleService.java 1597
SliderFieldPanel.java 348
SliderFieldPanelBeanInfo.java 364
SliderFieldPanelCustomizer.java 376
SnapshotUsage.java 1281
SplashScreen.java 317
StoppableChatServer.java 811
styles.css 663, 666
Survey.html 582
SurveyServlet.java 578
systemclock.idl 1443
SystemClockClient.java 1449, 1511
SystemClockImpl.java 1445
TakeOperation.java 1273
TallyDisplay.java 975
TallyPanel.java 950, 977
TintImage.cpp 1724
TipTestMIDlet Tip Test answer screen. 780
TipTestMIDlet.java 762
TipTestServlet.java 726
TitlesBean.java 663, 667, 669
toc.html 607
TransactionManagerFinder.java 1287
TransferableShape.java 285
Transform.java 1677
UnicastDiscovery.java 1204
UnicastDiscoveryUtility.java 1232
UnicastSeminarInfoClient.java 1223
UpdateCartServlet.java 1038
UpdateInputWindow.java 1285
UpdateOperation.java 1289
usage.xsl 1684
usage1.xsl 1691
usage2.xsl 1690
variables.xsl 1695
viewCart.jsp 663, 664, 686
ViewCartServlet.java 1026
ViewOrderHistoryServlet.java 1061
ViewOrderServlet.java 1065
VoteCollector.java 945
VoteCollectorEJB.java 972
VoteListener.java 949
Voter.java 941
WAPTipQuestion.xsl 746
WeatherBean.java 799
WeatherCellRenderer.java 805
WeatherDisplay.java 965
WeatherItem.java 805
WeatherListener.java 964
WeatherListModel.java 803
WeatherPublisher.java 953
WeatherService.java 793, 1602
WeatherServiceClient.java 801, 1605
WeatherServiceImpl.java 794
WeatherSubscriber.java 958
WebBrowser.java 38, 67
WebBrowserPane.java 31
weblogic.xml 1190
weblogic-cmp-rdbms-jar-address.xml 1178
weblogic-cmp-rdbms-jar-Customer.xml 1180
weblogic-cmp-rdbms-jar-order.xml 1184
weblogic-ejb-jar.xml 1168
WebToolBar.java 35, 63
welcome.jsp 601
welcome2.xml 1637
Welcome2TagHandler.java 645
WelcomeServlet.html 541, 554
WelcomeServlet.java 540, 720
WelcomeServlet2.html 551
WelcomeServlet2.java 549
WelcomeServlet3.java 553
WelcomeTagHandler.java 641
Whitespace characters in an XML document 1616
whitespace.xml 1616, 1646
FrontEndSpace tab in StartService GUI 1262
FULL date format 606
fully qualified name 459
Futurestep.com 1746

G
games.xml 1681
Gamma, Erich 20
“gang of four” 20, 24
General 918
general entity 1639
General Inter-ORB Protocol (GIOP) 1440
GeneralDiscoveryUtility.java 1237
GeneralPath 138, 144, 159, 218
Generate Default SQL 981
Generate SQL Now 1155
generate XHTML 667
generateSecret method from SecretKeyFactory 404
generic identifier 1630
GenericServlet class 535
Geometry 161, 195
gesture 56
get method 346, 1324
get request 533, 536, 539, 541, 549, 553, 556, 561, 564, 569, 601, 720, 751, 755
getAllFrames method of class JDesktopPane 316
getAllowsChildren method of class DefaultTreeModel 117
GetAllProductsServlet.java 1043
GetArrayElements function of C table JNIEnv 1719
getAddress method 1659
getAddress method of interface HttpServletRequest 573
getAddressNames method of interface HttpServletRequest 573
getAttributes method 1659
getAutoCommit method of interface Connection 482
getBeanDescriptor method of interface BeanInfo 369
getBundle method of class ResourceBundle 65
getBytes method from String 404
getchildCount method of interface TreeModel 127
getchildNodes method 1658, 1659
columnClassName method of interface ResultSetMetaData 474
columnCount method of interface ResultSetMetaData 465, 468, 475
columnCount method of interface TreeModel 111, 470, 475
columnName method of interface ResultSetMetaData 475
columnName method of interface TreeModel 111, 470, 475
columnType method of interface ResultSetMetaData 468
comment method of class Cookie 568
getConnection method of class DriverManager 465, 467
collection method of interface PooledConnection 519
collectionPane method of class JScrollPane 56
collection method of interface HttpSession 469
collection method of class HyperlinkEvent 37
getFirstID function of C table JNIEnv 1716
getGroups method of interface ServiceRegistrar 1208
getHandle method of interface
getHandle 848
getHomeHandle method of interface EJBObject 849
getAddress method of class Look-upLocator 1208
getIcon method of interface BeanInfo 369
getID method of interface HttpSession 573
getIndexOfChild method of interface TreeModel 128
getInt method of interface ResultSet 469
getLease method of interface ServiceRegistration 1247
getLastSelectedPathComponent method of class TreePath 128
getLastSelectedPathComponent method of class JTree 123
getLease method of interface ServiceRegistration 1380, 1389
getLookupEntries method 1378, 1379
getmaxAge method of class Cookie 568
getMaxAscent method of class FontMetrics 101
getMaxInactiveInterval method of interface HttpSession 573
GetMessage.java 1599
getMetaData method of interface ResultSet 465
getMethodDescriptors method of interface BeanInfo 369
GetMethodID function of C table JNIEnv 1713
getMoreResults method of interface Statement 517, 518
getName method of class Cookie 568
getNextSibling method 1659
getNodeName method 1659
getNodeType method 1659
getNodeValue method 1659
getObject method of interface ResultSet 469, 475
GetObjectArrayElements function of C table JNIEnv 1721
GetObjectClass method of interface C table JNIEnv 1708
getObject method of interface HttpServletRequest 539, 541
getParagraph method of interface HttpServletRequest 537, 538, 550, 561
getPasswordHintServlet.java 1068
getPassword method of class Cookie 568
getPooledConnection method of interface ConnectionPoolDataSource 519
getPort method of class LookupLocator 1208
getPreferredConfiguration method 167, 180
getPreferredSize method of class JPanel 96
GetProductServlet.java 1046
getPropertyDescriptors method of interface BeanInfo 369
getQueryString method of interface HttpServletRequest 584
getRaster 155
getRegistrar method of class LookupLocator 1208, 1227
getRegistrars method of class DiscoveryEvent 1209
getRequestDispatcher method of interface ServletContext 679
getRequestDispatcher method of interface ServletRequest 677, 679
getRequestDispatcher method of interface ServletRequest 679
getRequestURL method of class HttpServletRequest 584
getResource method of interface ServletContext 1011
getResultSet method of interface Statement 517, 518
getRow method of interface ResultSet 476
getRowCount method of interface TableModel 111, 470, 475
getScaleInstance static method of class AffineTransform 260
getSecure method of class Cookie 568
getSelectedNode method of class JTree 122
getSelectedValue method of class JList 111
getServletConfig method of interface Servlet 535
getServletContext method of interface ServletContext 678
getServletInfo method of interface Servlet 535
getSession method of interface HttpServletRequest 538, 572, 573
GetStaticFieldID function of C table JNIEnv 1716
GetStaticMethodID function of C table JNIEnv 1716
GetStaticReturnType function of C table JNIEnv 1716
getString method of class ResourceBundle 66
getStringUTFChars function of C table JNIEnv 1708
getTagName method of getter method 1323, 1324
getTransferable method of class DropTargetDropEvent 57
getTransferData method of interface Transferable 61, 284, 287
getTransferDataFlavors method of interface Transferable 287
GettypeField function of C table JNIEnv 1716
getUpdateCount method of interface Statement 517, 518
getUpdateCounts method of class BatchUpdateException 517
getValue method of class Cookie 565, 568
getValueAt method of interface TableModel 112, 470, 475
getVariableInfo method of class TagExtraInfo 651
getVersion method of class Cookie 568
getWriter method of interface HttpServletRequest 539, 541
getWriter method of interface HttpServletResponse 724
glyph 1765
Gnutella 16, 1549, 1550, 1551, 1590
goal detection 196
GoalDetector.java 205
gradient 139, 142, 218
GradientIcon.java 307
GradientPaint class 221
Grad, Mark 25
graphic 136, 138, 159
graphical programming and design environments 9, 322
Graphics 139, 142, 150
Graphics class 101
graphics context 136, 212
graphics primitive 139
graphics programming 8, 160, 212
Graphics2D 139, 142, 143, 146, 149, 212, 222
GraphicsConfiguration 167, 180
grayscale 147
GridLayout class 361
Group 167, 180
group-based access control (GBAC) 418
GROUP BY 453
group discovery 1209
GuestBean.java 623
guestBookErrorPage.jsp 631
guestBookLogin.jsp 626
GuestBookTag.java 649
GuestBookTagExtraInfo.java 652
guestBookView.jsp 629
GuestDataBean.java 624
GUI 160
GUI component 339, 347
GUI JavaBean 325

H
hacker 387
Handle interface 849
handleNotification method of interface NotificationListener 1343
handshake point 542
hardware 160
hardware independence 160
hasChildNodes method 1659
hash function 406
hash value 406, 407
hashCode method of class Object 1111
HashMap class 96, 686, 1582
Hashtable 155
hashtable 23
header 340, 342, 560
header element 1596
header file 1706, 1707
Headhunter.net 1748
height attribute of <jsp:plugin> action 614
Hellman, Martin 390
Helm, Richard 20
helper utilities 1232
heterogeneous network 15, 1366
high-level 160
Hire.com 1746
HireAbility.com 1750
Hirediversity.com 1745
Home Interface 865, 984
home interface 847, 848, 853
“hook up” an event 322
horizontal CORBAfacilities 1457
HORIZONTAL_SPLIT constant of class JSplitPane 48
host 798
host name 542
hot deployment 1164
HotDispatch.com 1750
HotJobs.com 1743, 1748
hspace attribute of <jsp:plugin> action 614
HTML 534, 539, 552
HTML document 539
HTML scraping 797
HTTP 995
HTTP (HyperText Transfer Protocol) 534, 536, 560, 1551
http authentication information 536
HTTP header 564
HTTP request 536
HTTP request type 536
HTTP server 837, 838, 1571
http.port 1700
http://www.w3.org/1999/XSL/Transform 1679
HttpConnection class 774, 775, 778
HttpServletRequest (javax.servlet.http) 599
HttpServletRequest class 722, 744, 1023
HttpServletRequest interface 536, 537, 538, 539, 553
HttpServletRequest interface from javax.servlet.http 537, 572
HttpServletRequest (javax.servlet.http) 599
HttpServletRequest class 724, 739, 1019
HttpServletRequest interface 536, 538, 539, 540, 553
HttpServletRequest interface from javax.servlet.http 538
HttpUtils class 584
hyperlink 31
HyperlinkEvent class 37
HyperlinkEvent.EventType.ACTIVATED
ACTIVATED constant of class HyperlinkEvent.EventType 37
HyperlinkListener interface 37
hyperlinkUpdate method of interface HyperlinkListener 37
HyperText Transfer Protocol (HTTP) 10

I
IBM Corporation 1764
IBM WebSphere 1163, 1165
IBM WebSphere application server 534
Index

Icon interface 45, 307
iconify 52
ID 1639, 1640, 1641
id 1638, 1640
id attribute of <jsp:useBean> action 618
IDE 340, 370
IDExample.xml 1639, 1641
IDL (interface definition language) 791
idlj IDL compiler 1443, 1445, 1452, 1458, 1459, 1493, 1496, 1498
IDREF 1638, 1639, 1641
idref 1638
iepluginurl attribute of <jsp:plugin> action 614
if element 1687
Igniter GUI 1408, 1409, 1426
Igniter.bat 1367
IGNORE 1644
IllegalArgumentException class 90, 91, 848, 1083, 1390, 1391
IllegalStateException class 917
illuminate 161
Image class 149, 155, 168, 778
image processing 136, 139, 146, 150, 159, 218
Image Processor case study 1295
ImageDisplayer.java 1310
ImageEntry.java 1295
ImageIcon class 43, 339, 1299
ImagePanel 147
ImagePanel.java 147
ImageParser.java 1309
ImageProcessor.java 1295
ImageProcessorMain.java 1301
ImageSeparator.java 1301
ImageSizeException.java 1722
IMConstants.java 1575
i-mode 719, 785
IMODETipAnswer.xsl 754
IMODETipQuestion.xsl 751
IMPeer.java 1553
IMPeerImpl.java 1560
IMPeerListener.java 1557
Implementation Class Name 1192
implementation JAR file 1394
implementation transparency 1437, 1501
implicit object 598
implicit object scope 598
#IMPLIED 1637, 1639, 1641, 1643
import attribute of page directive 629, 634, 635
import element 1690, 1691
IMService.java 1553
IMServiceImpl.java 1555
IMServiceManager.java 1562
incarnation number 819
!INCLUDE 1644
INCLUDE 1644
include a resource 604
include content from a resource 679
include directive 605, 634, 636
include element 1692
include method of class RequestDispatcher 679
include.jsp 608
includeDirective.jsp 637
index.html 663, 665
indexed property 355, 357
Individual Expert Participation Agreement (IEPA) 1702
industry.java.sun.com/products/jdbc/drivers 521
infinity symbol 452
info attribute of page directive 635
info element 642, 646, 652
information tier 26, 992, 993
information-tier 717, 905
Informix 446, 695
init method of class Cipher 404
init method of interface Servlet 535, 581
init-param element 581
InitialContext class 667, 669, 854
initialization parameter 581
input-processing component 86
InputStream class 778, 1011
INSERT SQL statement 453, 460, 498, 906, 920, 1659
insertNodeInto method of class DefaultTreeModel 122
insertRow method of interface ResultSet 518
insignificant whitespace 1617
Install JavaBean dialog 323
InstanceNotFoundException class 1332
Instant Messenger 1550
instantiatate 375
Instrumentation 1322
instrumentation level 15, 1321
integrated development environment 323, 340
integrity 388, 389, 406
intellectual-property law 1551
intelligent agent 1740
InterestCalculator.java 850
InterestCalculatorClient.java 854
InterestCalculatorEJB.java 852
InterestCalculatorHome.java 851
Interface Definition Language (IDL) 1438, 1439, 1440, 1441, 1443, 1444, 1448, 1449, 1452, 1454, 1455, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1467, 1468, 1474, 1476, 1477, 1481, 1483, 1489, 1492, 1493, 1495, 1496, 1497, 1499, 1501, 1503, 1504, 1509, 1510, 1514, 1515, 1516, 1519, 1521, 1522, 1523, 1527, 1528, 1530, 1531, 1543
interface definition language (IDL) 791
interface JAR file 1394
interface Node 1658
interface Remote 793
internal subset 1629
internalFrameActivated method of interface InternalFrameListener 316
internalFrameClosing method of interface InternalFrameListener 316
internationalization 62, 68
internationalized 1389
internationalized application 30
Internet 531
Internet Explorer 539
Internet Information Server (IIS) 543
Internet Inter-ORB Protocol (IIOP) 860, 1438, 1440, 1499, 1501
Internet Policy Registration Authority (IPRA) 407
Internshipprograms.com 1753
interoperability 16, 1439, 1440, 1500, 1595
Interoperable Object Reference (IOR) 1440, 1451, 1481, 1484, 1501, 1504
Interpolator 180, 193, 195, 196
InterruptedException class 1270
InterviewSmart 1753
intro-invalid.xml 1631
intro.xml 1664, 1680
intro.xsl 115
intro2.xml 120
introspection 346, 364
IntrospectionException class 370, 371
InvalidAlgorithmParameterException class 404
invalidate method of interface HttpSession 573
invalidate method of interface HttpSession 693
invalidate method of interface HttpSession 573
InvalidKeyException class 404
InvalidKeySpecExceptions class 404
InvertFilter.java 152
invocation transparency 1437, 1501
invoke method from MBeanServer 1332
invoke method of class Call 1604
invoke method of class MBeanServer 1332
invokeLater method of class SwingUtilities 835, 1209, 1241
IOException class 33, 284
iPlanet Application Server 991, 1163, 1164
isCellEditable method of interface TableModel 112
isDataFlavorSupported method of interface Transferable 61, 287
isErrorPage attribute of page directive 631, 636
isIdentical method of interface EJBObject 848
IsInstanceOf function of C table JNIEnv 1727
isNew method of interface HttpSession 573
ISO Country Code 71
ISO Language Code 71
isThreadSafe attribute of page directive 635
ItemListener 156, 178
Iterator class 361, 1583
Iterator design pattern 23
Iterator interface 257, 674, 676, 1100
J
J2EE 17
J2EE 1.2.1 reference implementation 11
J2EE application server 847, 1153
J2EE-compliant application server 1153
J2EE config directory 695
J2EE specification 1164
J2EE_CLASSPATH 1700
J2EE_HOME environment variable 1699
J2ME 11, 719, 755, 756, 758, 762, 785, 786
Jakarta project 533
jakarta-tomcat-3.2.1 543
jakarta.apache.org 533
jakarta.apache.org/jetty2
jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.3/bin/543
JAR (Java archive) file 340, 342, 343, 346, 353, 361, 378, 1597
Java Authentication and Authorization Service (JAAS) 10, 388, 417, 418, 422
Java Bean header of manifest file 342, 344
Java Community Process 533, 1366
Java Community Processes (JCP) 17
Java Cryptography Extension (JCE) 10, 388, 395
Java Cryptography Extensions (JCE) 6
Java Database Connectivity (JDBC) 10
Java Database Connectivity (JDBC) API 10, 446
Java Dynamic Management Kit (JDMK) 1322
Java extension mechanism 679
Java Foundation Classes (JFC) 8, 30
Java Interface Definition Language (JavaIDL) 15
Java Look and Feel Graphics Repository 510
Java Management Extensions 15
Java Management Extensions (JMX) 1321, 1322, 1323, 1331, 1338
Java Message Service (JMS) 939, 1259
Java Naming and Directory Interface (JNDI) 669, 1534, 1542
Java Native Interface 17
Java Native Interface (JNI) 17, 759, 1706
Java-object-to-Java-object distributed communication 791
Java Plug-in 604, 613, 616
Java properties file 62
Java Remote Method Protocol (JRMP) 1530
Java Sandbox 410, 413
Java Secure Socket Extension (JSSE) 10, 424
Java Secure Sockets Extensions (JSSE) 6
Java Server Pages (JSP) 11
Java Server Pages 1.1 specification 636
Java servlet 847
Java Servlet API 10
Java Specification Request (JSR) 1702, 1703, 1704
Java Transaction API 879
Java Virtual Machine 429
java.awt package 137, 138
java.awt.color package 138
java.awt.datatransfer package 56
java.awt.font package 138
java.awt.geom package 138, 140
java.awt.image package 138
java.awt.image.renderable package 138
java.awt.print package 138
java.beans package 347
java.io package 758
java.lang package 758
java.lang.ClassNotFoundException 466
java.lang.Cloneable 22
java.lang.reflect package 364
java.net package 531, 532
java.rmi package 532, 792, 798, 1266, 1276
java.rmi.activation package 809
java.rmi.server package 796
java.rmi.server.codebase property 837, 1229
java.servlet.jsp.tagext package 641
java.sql package 464, 466, 468
java.sun.com/beans 323
java.sun.com/docs/ books/tutorial/jdbc/index.html 521
java.sun.com/j2ee 653
java.sun.com/j2se/ j2sdk1/techdocs/ api/javax/servlet/http/HttpServletRequest.html 537
java.sun.com/j2se/ j2sdk1/techdocs/ api/javax/servlet/http/HttpServletResponse.html 538
java.sun.com/j2se/1.3/ docs/guide/jdbc/getstart/GettingStartedTOC.fm.html 469
java.sun.com/j2se/1.3/ docs/guide/jdbc/index.html 521
java.sun.com/products/ jdbc 467, 520
java.sun.com/products/ jdbc/articles/package2.html 521
java.sun.com/products/ jdbc/faq.html 521
java.sun.com/products/ jsp 653
java.sun.com/products/ jsp/download.html 594, 653
java.sun.com/products/ servlet 653
java.sun.com/xml/down- load.htm 679
java.text package 234
java.util package 758, 776
java:comp/env naming context 918
JAVA_HOME environment variable 543, 1699
Java2DExample.java 156
Java2DImageFilter.java 150
Java3DExample.java 179
Java3DWorld.java 163
Java3DWorld1.java 181
JavaBean 9, 11, 322, 594, 670, 694
JavaBeans 9, 379
JavaBeans 1321
JavaBeans component 379
JavaBeans component architecture 9
JavaBeans design pattern 352, 364
JavaBeans documentation 379
JavaBeans home page 379
JavaBeans specification 379
javac 1571
javaFileListFlavor constant of class DataFlavor 61
javah JNI utility 1706, 1707, 1708, 1711, 1714
Java-object-to-Java-object distributed communication 12
JavaSpaces 14
JavaSpaces implementation outrigger 1261
JavaSpaces interface 1264
JavaSpaces operations read 1264
take 1264
write 1264, 1266
JavaSpaces.NO_WAIT constant 1272
JavaSpacesFinder.java 1262
JavaSpaces 14
JavaSpaces service 1259, 1299
persistent JavaSpaces service 1261
temporary JavaSpaces service 1261
JavaSpaces service 1260
JavaSpaces service operations
read 1269
take 1269
JavaSpaces service properties
JavaSpaces technology 1198
javax.ejb package 848
javax.fma.service package 1380
javax.fma.service.log package 1389, 1390
javax.fma.service.event package 1382
javax.fma.util package 1390
javax.jms package 940, 944
javax.management package 1331
javax.microedition.io package 759
javax.microedition.lcdui package 759
javax.microedition.midlet package 759, 761
javax.microedition.rms package 759
javax.naming package 667, 669, 918
javax.servlet package 532, 535, 539, 594
javax.servlet.http package 532, 535, 539, 564, 594
javax.servlet.jsp package 532, 594, 599
javax.servlet.jsp.tagext package 594, 638, 651
javax.sql package 519, 533, 661, 898, 905
javax.swing.table package 470
javax.xml.parsers package 676, 680, 1656
javax.xml.transform package 676
javax.xml.transform.dom package 676
javax.xml.transform.stream package 676
JAXP 1677
JAXP API (Application Programming Interface) 1654
jaxp.jar 679
jboolean JNI type 1708, 1719
JBoss 1163, 1165
JButton class 39, 44, 87, 88, 331
JCheckBox class 272, 944
JChooserPanel class 1301
jclass JNI type 1712, 1716, 1724
JComboBox class 68, 303
JComponent class 137
JDBC 11, 12, 466, 479, 515, 518, 519, 533, 661, 898, 905
JDBC (Java Database Connectivity) 10, 446, 464, 533, 577, 578
JDBC 2.0 519
JDBC data source 669
JDBC-database driver 726
JDBC driver 10, 446, 467
JDBC driver type 467
JDBC-Net pure Java driver (Type 3) 466
JDBC-to-ODBC bridge driver (Type 1) 466, 467
JDBC Web site 467
jdbc.datasources 1700
jdbc.datasources J2EE configuration property 695
jdbc.datasources property 906
jdbc.drivers 1700
jdbc:cloud:space:rmi:books 465, 467
jdbc:cloud:space:rmi:books JDBC URL 695
JDbcOdbcDriver 467
JDBCRowSet class 520
JDesktopPane class 52, 56, 316
JDialog class 304
JDK (Java Development Kit) 1322
jdmkrt.jar 1357
jdmktk.jar 1357
jdouble JNI type 1708, 1711
JEditorPane class 30, 31, 33, 404
JEditorPane class 404
jfieldID JNI type 1716
JFileChooser class 303
JFrame class 44, 52, 325, 396
Jigsaw Web server 534
Jini 14, 16, 1549, 1551, 1552, 1572, 1583, 1589
Jini CLASSPATH 1199
Jini core packages
net.jini.core.discovery package 1204
net.jini.core.entry 1220, 1260, 1270
net.jini.core.event 1276
net.jini.core.lookup 1204, 1220, 1228, 1262
net.jini.core.transaction 1266
Jini extension packages
net.jini.discovery 1209
net.jini.lease 1266
net.jini.lookup.entry 1220, 1261
net.jini.space 1264
Jini groups 1201
Jini lookup service 1198, 1204, 1369, 1378, 1381, 1395, 1399, 1434, 1552
Jini required services
lookup service 1199
rmid 1199
Web server 1199
Jini service proxy 1552
Jini services
JavaSpaces service 1299
lookup service 1261
transaction service 1261, 1287, 1299
Jini Software Kit 1198
Jini Technology Core Platform 1198
Jini Technology Core Platform Compatibility Kit 1198
Jini Technology Extended Platform 1198
Jini Technology Starter Kit 1198
Jini transaction manager 14, 1260, 1284
JINI-1.1-G-CS.zip 1198
jini11_win32.properties 1200
jini-core.jar 1198, 1269, 1272, 1273, 1280
jini-ext.jar 1198, 1269, 1272, 1273, 1280
jint JNI type 1708
jintArray JNI type 1718, 1719
JInternalFrame class 52, 56, 287, 316
JInternalFrame java 1731
JNIImageFrame.java 1710
JNIMethodNameWrapper.java
JNIPanel.java 1727
JNIPIMain.java 1714
JNIPIMethodWrapper.h
JNIPIWrapperImpl.java 1711
JNIPIWrapperImpl.java 1712
JNIPrintMain.java 1709
JNIPrintWrapper.java
JNIPrintWrapperImpl.cpp 1708
JNIStaticMain.java 1717
JNIStaticWrapper.java 1714
JNIStaticWrapperImpl.cpp 1716
JNIWhiteSpaceCore.java
JNIWhiteSpaceCoreImpl.cpp
jsp:forward> action 604
jsp:getProperty> action 605, 619
jsp:include> action 604, 605, 606, 636
jsp:plugin> action 604, 613
jsp:property> action 605, 621, 628, 629
jsp:useBean> action 604, 617, 629
jsp:usebean> action 639
jspException> method 595
jspInit> method 595
jspversion> element 642
JspWriter (package java-
as.servlet.jsp) 599
JspWriter class 642, 651
JSR Approval Ballot 1703
JSR Review 1703
Jstring JNI type 1708, 1709
JTabbedPane class 30, 45, 48
JToolBar class 33, 34, 37, 40, 44
JTree class 8, 86, 111
JTextArea class 31
JTextComponent class 31
JustCJobs.com 1750
JustComputerJobs.com 1750
JustJavaJobs.com 1739, 1750
K
K Virtual Machine 11
KaZaA 1550
Kerberos 417
Kernel 138, 152, 153
key 394
key agreement protocol 393
key algorithms 409
key distribution center 389
key exchange 389
key generation 394
key length 389
key management 393, 394
key theft 394
Keymap interface 829
keystore 409, 415, 424, 428
KeyStroke class 829
keytool utility 409, 414, 428
KVM 758, 760

L
LAN-based discovery 1590
language attribute of page directive 635
last auto-increment value 498
last method of interface ResultSet 476
Latin World 1746
launch Cloudscape database server 695
launch J2EE server 695
layer 26
layout 326
LayoutManager interface 96
LDAP 1165
Lea, Doug 25
Leaf 167
Lease interface 1247, 1266, 1410
lease utilities 1244
LeaseRenewalManager class 1244
Lease.FOREVER constant 1247, 1266, 1302, 1410
LeaseListener interface 1247
LeaseRenewalManager class 1244, 1247, 1249, 1410
least-recently used policy 853
left angle bracket ( < ) 1617
legacy system 1436, 1437, 1438, 1439, 1441, 1442, 1454, 1480, 1499
library 1706
library JAR file 694
life cycle 14
Light 161, 162
light 162, 195
light source 136
lighting 169, 212
lightweight component 8, 30
Lightweight Directory Access Protocol 1165
LIKE 455, 456, 458
Line2D 138
Line2D.Double class 140, 144
Line2D.Float class 222
linked list 23
Linux 1322
List class 96, 651, 772
List interface 61
List.EXCLUSIVE constant 774
List.IMPLICIT constant 774
List.MULTIPLE constant 774
ListCellRenderer interface 805
ListDataListener interface 107
ListModel interface 107, 803
live-code approach 2
load balancing 1164
loadLibrary method 1723
Local/Remote Interface 985
Locale 162
locale 62
Locale.FRANCE constant 68
Locale.US constant 68
localhost(127.0.0.1) 542,
544, 798
localizable message 1390
LocalizableMessage class 1390, 1392
localization 1763
LocalizationError class 1391
locate a naming service 669
locate records in a database 454
LocateRegistry class 1575
location transparency 1437, 1501
locator discovery 1204
log message 1389
log message category 1389
log method of interface LogService 1389, 1390
log repository 1389
log service 1368, 1369, 1378,
1381, 1389, 1415, 1416
LoginServlet.java 1057
LogMessage class 1389, 1390
LogMessage.AUDIT constant 1390
LogMessage.DEBUG constant 1390
LogMessage.ERROR constant 1390
LogMessage.INFO constant 1390
LogMessage.TRACE constant 1390
LogMessage.WARNING constant 1390
LogoAnimator.java 337
LogoAnimator2.java 345
LogService interface 1390
LONG date format 606
LONG_DESCRIPTION constant of interface Action 45
lookup 11
lookup method of interface Context 669, 854
lookup method of interface ServiceRegistrar 1228
lookup service 1199, 1214, 1562,
1564, 1572
LookupBrowser panel 1203
LookupCache interface 1253
LookupDiscovery class 1209,
1220, 1232, 1237
LookupDiscoveryManager class 1237
LookupDiscoveryManager.FROM_GROUP
constant 1237
LookupDiscoveryManager
class 1249
LookupLocator class 1204,
1207, 1232, 1262, 1571
LookupLocatorDiscovery 1232, 1235
LookupLocatorDiscovery class 1232, 1237
LookupOp 151
low-level 160
LowTonerPolicy.java 1415
LowTonerPolicyImpl.java 1416
LowTonerPolicyImpl.properties 1420
Lucifer 390
Model-View-Controller (MVC) 8, 9
module CORBA keyword 1443, 1458, 1460, 1469, 1482, 1501, 1503
MoneyTransfer.java 880
MoneyTransferEJB.java 881, 887
MoneyTransferEJBClient.java 892
MoneyTransferHome.java 880
Monster.com 1739, 1743, 1748, 1751
MorganWorks.com 1746
morphing 161
motion 161
mouse gesture 56
MouseListener interface 265, 302
MouseMotionAdapter class 265
MouseMotionListener interface 265, 302
MouseRotate 169, 178
MouseScale 169, 178
MouseTranslate 169, 178
moveToCurrentRow method of interface ResultSet 519
moveToInsertRow method of interface ResultSet 519
Mozilla 1019
MSN Instant Messenger 1589
MSN.com 560
multit-byte character set (MBCS) 1766
multi-pooling 1164
multi-tier application 992
multicast announcement protocol 14
multicast discovery 1204, 1209, 1552
multicast group 1582
multicast request protocol 14, 1399
multicast socket 16, 1549, 1572
MulticastDiscovery.java 1209
MulticastPacket class 1576
MulticastReceivingThread.java 1577
MulticastSendingThread.java 1572
MulticastSocket class 1577
multimedia 531
multiple document interface 52, 287
Multipurpose Internet Mail Extension (MIME) type 723
multithreading 24, 25, 531
multitier architecture 578
multitier client-server application 533
multitier Web-based survey using XHTML, servlets and JDBC 578
MutableTreeNode class 117
MVC 13, 86, 87, 104, 991, 1010
MVC architecture 220, 319
MVC design pattern 995, 1075
MyAbstractAction.java 66
MyErrorHandler.java 1663
MyImage.java 240
MyLine.java 230
MyLineController.java 267
MyOval.java 234
MyRectangle.java 232
MyShape.java 224
MyShapeController.java 261
MyShapeControllerFactory.java 273
MySQL 446
MyText.java 235
MyTextController.java 269

N
name attribute 1683
name attribute of <jsp:param> action 610
name attribute of <jsp:plugin> action 614
name attribute of <jsp:set-Property> action 622
Name class 1562, 1570
NAME constant of interface Action 43, 45
name element 642, 646, 652
Name entry 1228, 1242, 1261
Name header of manifest file 342
name lookup 519, 669
name resolution 519, 669
name token 1643
name/value pair 604
namespace 430, 1620
namespace prefix 1620
Namespace prefix xsl 1679
Naming class 819
naming context 854, 1448
Naming Service 1447, 1448, 1449, 1452, 1456, 1514, 1516, 1525, 1527, 1529, 1534, 1535, 1538, 1539, 1542
naming service 519, 669
NamingException class 669, 670, 861, 918, 925, 1023, 1046
narrow method of class PortableRemoteObject 860
National Institute of Standards and Technology (NIST) 390
National Security Agency (NSA) 390
National Weather Service 792, 794, 798
native 17, 1706, 1709, 1710, 1712, 1714, 1716, 1717, 1718, 1719, 1722, 1723, 1736
Native-API, partly Java driver (Type 2) 466
native function 1706
native library 1706, 1707, 1709, 1710, 1711, 1717
native method 1706, 1707
Native-Protocol pure Java driver (Type 4) 466
navigate 196
navigation 179, 196, 197
Navigator.java 200
NDATA keyword 1641
NESTED constant 651
net.jini.core.discovery. package 1204
net.jini.core.entry package 1220, 1260, 1270
net.jini.core.event package 1276
net.jini.core.lookup package 1220, 1262
net.jini.core.transaction package 1266
net.jini.lease packages 1266
net.jini.lookup.entry package 1220, 1261
net.jini.space package 1264
network 160
network administrator 1409
network-based service 1198
network incompatibility 1589
network management 1320, 1321
network management developer 1321
network management solution 1321
network manager 1320
network security 387
network traffic 1551
Never transaction type 892
New Enterprise Bean 863, 978, 1149
New Enterprise Bean Wizard 898, 918
newInstance method of class Class 265
NewIntArray function of C table JNIEnv 1719
NewObject function of C table JNIEnv 1721
NewObjectArray function of C table JNIEnv 1719
newRepeatedDateSchedule method of interface SchedulingService 1391
newTransformer method of class TransformerFactory 1019
next method of interface ResultSet 465, 468
NMTOKEN 1638, 1643
no_point_objects 1395
Node 180, 195
Node interface 1658
Node.ATTRIBUTE_NODE constant 1659
Node.CDATA_SECTION_NODE constant 1659
Node.COMMENT_NODE constant 1659
Node.ELEMENT_NODE constant 1659
Node.PROCESSING_INSTRUCTION_NODE constant 1659
Node.TEXT_NODE constant 1659
NodeComponent 167
NodeList class 1019
NodeList interface 1658
non-modal dialog 304
non-repudiation 388, 406
nondurable subscription 951
non-validating parser 1628
NoSuchAlgorithmException class 404
NoSuchMethodException class 369
NoSuchPaddingException class 404
NOTATION 1644
notation 1641
notification 1343, 1382
notification broadcaster 1342
notification broadcaster MBean 1342
Notification Service 1521, 1522, 1527
notification type 1342
NotificationBroadcastBanner interface 1342
NotificationBroadcasterSupport class 1342
NotificationFilter interface 1343
NotificationListener interface 1343
notify method of interface JavaSpace 1265, 1276
notify method of interface RemoteEventListener 1381, 1392
notifyAll method of class Object 1399
notifyObservers method of class Observable 88, 89, 91, 93
NotifyOperation.java 1277
NotSupported transaction type 891
nspluginurl attribute of <jsp:plugin> action 614
n-tier application 992
NullPointerException class 96
NumberFormatException class 102, 104
Observer design pattern 8, 86, 88
Observer interface 88, 91, 93, 243, 254
observer listener 1381, 1406, 1428
occurrence indicator 1631
occurrence indicators 1632
ODBC data source 467
one-to-many relationship 452, 479, 1100
Online Certificate Status Protocol (OCSP) 408
online contracting services 1750
online recruiting 1741
online transactions 11
onMessage method of interface MessageListener 949, 974
opaque network reference 1440
Open Property File 1200
OpenGL 160, 161
openStream method of class URL 797
Openwave UP simulator 719, 720, 739, 785
options request 537
Oracle 446
ORDER BY 453, 456, 457, 458, 459
order.html 663, 664, 689
Order.java 1095
ordered 447
OrderEJB.java 1095
OrderHome.java 1101
ordering of records 453
OrderModel.java 1102
OrderProduct.java 1107
org.apache.soap.rpc package 1601
org.omg 532
org.omg.CORBA.IDLEntity 1444, 1495, 1504
org.xml.sax package 676
orientation property of class JToolBar 37
originator object 24
out implicit object 599
OutofPaperPolicy.java 1410
OutofPaperPolicyImpl.java 1411
OutofPaperPolicyImpl.properties 1415
output parameter for a CallableStatement 515
outrigger 1261
P
P2P (peer-to-peer) 16, 1549, 1550, 1551, 1552, 1572, 1588, 1589, 1590
package directory structure 701
package-based communication 532
page attribute of \texttt{<jsp:forward>} action 610
page attribute of \texttt{<jsp:include>} action 605
page directive 622, 628, 629, 634, 635
page implicit object 599
page scope 598, 617, 618, 628
PageContext class 647, 651
PageContext implicit object 599, 642
Paint 138, 144
paint 137, 139
Paint Shop Pro 147
paintComponent method of class JPanel 101
param attribute of \texttt{<jsp:setProperty>} action 622
Parameter class 1601
parameter entity 1645
parent element 1614
parent node 116
parentheses () 1630
parseDouble method of class Double 861
parsePostData method of class HttpUtils 584
parseQueryString method of class HttpUtils 584
ParserConfigurationException class 678
passivate 853
passivation 1525
Password-Based Encryption (PBE) 395
path mapping 547
pattern matching 455
pauseApp method of class MIDlet 761
PBE (Password-Based Encryption) 395
PBEKeySpec class 404
PBEParameterSpec class 404
PCDATA 1629, 1630, 1635, 1636, 1637, 1639, 1641, 1642, 1644, 1645
peer 1589
peer discovery 1551, 1589
peer groups 1589
peer-to-peer (P2P) 1549, 1550, 1551, 1552, 1572, 1588, 1589, 1590
peer-to-peer application 1549, 1550, 1551, 1552, 1572, 1589
peer-to-peer computing 1198
PeerDiscoveryListener.java 1582
PeerList.java 1564, 1583
peer-to-peer (P2P) 16
peer-to-peer application 16
Peoplescape.com 1746
percent (%) SQL wildcard character 455
percent character 1645
permission 410
persistence 1168, 1175, 1176
persistence-descr 1175, 1176
Persistence Management dialog 924
persistence-type 1175
persistence-use 1175
persistent data source 905
persistent information 560
persistent JavaSpace service 1261
Persistent State Service (PSS) 1519, 1520, 1526
personalization 559, 560
personalized content 1053
PGP 392
phase delay 194
PhilosophersJList.java 108
PhilosophersJTable.java 112
PhilosophersJTree.java 117
PIContainer.java 1711
pie-chart view 96
pipe 1589
pipe character (|) 1631, 1643
pipeline 139
pixel 147, 151, 155
pixel color value 151
PixelTintException.java 1722
Pixo i-mode browser 739
plaintext 388, 393
platform incompatibility 1589
Platform-Independent Model (PIM) 1543
plug-and-play 1198
Pluggable Authentication Module (PAM) 418
pluggable look and feel 8, 30
plugin.jsp 616
plus sign (+) 1631, 1632, 1636
PNG 339, 340, 344
Point class 284
point object 1379
point-to-point connection 424
point-to-point messaging model 938, 940
Point2D class 233
Point3f 195
PoliciesStarter.java 1424
policy 1208, 1261
policy creation authorities 407
policy file 410, 412, 1426
policy.all 1208
policytool 410
polymorphism 221
PooledConnection interface 519
pooling 849
pool-name 1178
port 543
port 80 542
port 8080 542
port number 542, 797, 798
portability 137, 160, 1766
Portable Object Adapter (POA) 1456, 1481, 1499, 1502, 1514, 1515, 1516, 1519, 1524, 1525, 1529, 1543, 1544
PortableRemoteObject class 860
position checking 179, 196, 197
position information 168
PositionInterpolator 194
post method from EventService 1383
post request 533, 536, 538, 539, 552, 554, 508, 564, 569, 574, 578, 720, 744
predefined software component 322
predefined software components 9
prefix attribute of taglib directive 640
prepareCall method of interface ResultSet 518
PreparedStatement class 625, 886, 916
private key 390, 392, 394, 406, 407, 1572
process.jsp 663, 664, 693
ProcessingInstruction interface 1658
Product.java 1085
ProductDetails.xsl 996
ProductEJB.java 1086
ProductHome.java 1089
ProductModel.java 1089
ProductNotFoundException 1075
ProductSearchServlet.java 1050
Program Management Office (PMO) 1701
programmer-defined event 355
promotion 559
properties file 68, 71, 1393, 1416
property attribute of <jsp:setProperty> action 622, 628
property file for configuring the StartService tool 1200
property get method 346
property set method 346
property sheet 371, 375
propertyChange method of interface PropertyChangeListener 353, 371
PropertyChangeEvent class 347, 351, 361, 378
PropertyChangeListener interface 347, 352, 361, 378
PropertyChangeSupport class 347, 351
PropertyDescriptor class 369, 370
PropertyEditor interface 370, 371, 374
PropertyEditorSupport class 371
proprietary protocol 1320
protocol 393, 534
protocol adaptor 1338
protocol connector 1338
protocol for communication (jdbc) 468
Prototype design pattern 22
provider-based architecture 395
proxy 1392, 1395, 1410, 1421, 1439, 1447, 1448, 1453, 1463, 1465, 1492, 1503
PUBLIC 1629
public key 390, 392, 407, 1572
public-key algorithm 390, 393, 394
public-key cryptography 390, 392, 393, 394, 407, 409
Public-key Infrastructure (PKI) 407, 409
publish 938, 951, 957, 958
publish-subscribe event system 1381
publish/subscribe messaging model 938, 951
publishers table of books database 448
pull model application 1468, 1521
push model application 1468, 1521
PUSH_MODE 1343
put request 537
putValue method of interface Action 43, 51, 56, 65, 72
Python 1616

Q
Quality of Service (QoS) 1466, 1481, 1516, 1529, 1530
query 446, 447
query a database 464
query application for the Books.mdb database 528
question mark (?) 1631, 1632, 1633, 1636
Queue class 944
QueueConnection class 944
QueueConnectionFactory class 944
QueueReceiver class 944
QueueSender class 944
QueueSession class 944

R
Random object 740
Raster 138, 147, 150, 155
RasterOp 139, 150, 151, 155
RDBMS 1176
read method of interface JavaSpace 1264, 1269, 1272
read-only 155
read operation 1259
Read/Write Lock design pattern 25
read/write property 346, 670
readIfExists method of interface JavaSpace 1264, 1269
ReadOperation.java 1272
read-timeout-seconds 1168
real-time search 1550, 1551
rebind method of class Naming 798
record 446, 447, 451, 453, 454, 455, 456, 457, 459, 460, 461, 479, 523
record set 447
Recruitsoft.com 1746
Rectangle2D.Double class 140
Rectangle2D.Float class 232, 233, 242
RectangularShape class 138
redirect a request 556
redirecting requests to other resources 556
RedirectServlet.html 558
RedirectServlet.java 556
Refer.com 1746
reference-descriptor 1176, 1190
Reference Implementation (RI) 1703, 1704
ReflectionException class 1332
refresh interval 598
Reggie panel 1201
register a data source with a naming server 695
register method of interface ServiceRegistrar 1223
RegisterServlet.java 1053
Registrar menu 1203
registry for remote objects 798
rejectDrag method of class DropTargetDragEvent 62
rejectDrop method of class DropTargetDropEvent 61, 284
relational database 10, 446
relational database management system (RDBMS) 992
relational database model 446
relational database table 446
ReleaseStringUTFChars function of C table JNIEnv 1709
Re-leasertypeArrayElements function of C table JNIEnv 1719
remote client 1369
Remote Home Interface 978
Remote Interface 865
Remote interface 792, 793, 810, 847, 1554
Remote Interfaces 978
remote method 792
remote method call 532, 791, 793
Remote Method Innovation (RMI) 791
Remote Method Invocation (RMI) 11, 16, 532, 1480, 1495, 1498, 1499, 1504, 1528, 1529, 1530, 1549, 1552, 1571, 1575, 1577, 1588
remote object 12, 791, 792, 796
remote object class 807
remote object implementation 792, 793
remote procedure call (RPC) 791, 1596
remote reference 792, 1028, 1552
remote server object name 798
RemoteEventListener interface 1276, 1379, 1382, 1392
RemoteEventListenerImpl class 1399
RemoteException class 793, 797, 848, 873, 906, 925, 1075, 1266, 1270, 1570
remotely accessible 12, 791
remove method 847
remove method of interface EJBObject 848
remove methods of Entity EJB 906
removeAttribute method 1660
removeChild method 1659
RemoveException class 862
RemoveFromCartServlet.java 1036
removeRow method of class DefaultTableModel 115
removeTableModelListener method of interface TableModel 111, 470
render 139, 180, 196
Index

rendering an XHTML document 666
RenderingHints 151
rendezvous discovery 1590
renewUntil method of class LeaseRenewalManager 1247
repaint 137, 150
repaint method of class JPanel 257
replaceChild method 1659
request for proposal 1751
Request for Proposal (RFP) 1523
request implicit object 599
request method 536
request parameter 559
request scope 598, 617, 618, 628
request-time error 595
request type 536
RequestDispatcher class 684
RequestDispatcher interface 677, 679, 684
#REQUIRED 1637, 1638, 1639, 1641, 1642, 1643
required element of tag library descriptor 646
Required transaction type 891
RequiresNew transaction type 891
RescaleOp 151
resolve method of interface URIResolver 1011
resource file 1390
Resource References dialog 918
resource.properties file 978
ResourceBundle class 62, 63, 65, 66
response 600
Response class 1604
response implicit object 599
responsible listener 1381, 1428
restricted algorithm 389
result set 447, 454
result set concurrency 473
result set type 473
ResultSet column name 469
ResultSet column number 469
ResultSet concurrency constant 475
ResultSet interface 111, 465, 468, 473, 475, 476, 516, 520, 670, 739, 917
ResultSet method 625
ResultSet type 474
ResultSetConcurrency 475
ResultSetMetaData interface 465, 468, 474, 475
ResultSetTableModel.java 470
resume 1740, 1745, 1748
resume-filtering software 1745
Return Client JAR 986
Return Client Jar 868
revoked certificate 408
RGB 147
Rijndael 390
Rivest, Ron 392
RMI 669, 1198, 1602
RMI activation daemon 809, 816, 836, 1199, 1571
RMI activation mechanism 811
RMI activation system 1366
RMI callback 810, 820
RMI connections 1339
RMI connection 1322, 1339
RMI connection client 1343
RMI connector 1346, 1347, 1356
RMID panel 1201
RMI utility 809, 836, 1199, 1203, 1261
RMIIOPMessageManager.java 1539
RmiJdbc Cloudscape framework 463
RMIMessageManager.java 823
rmiregistry utility 807, 815, 837
RMIRegistry class 835, 1208
RMI SecurityManager class 835, 1208
Role-based access control (RBAC) 418
rollback 482, 498
rollback method of interface Connection 482, 516
root certification authority 407
Root Directory 863
Root Directory 1149
root element 1614
root key 407
root node 116
rotate 136, 163, 180, 212
rotation 168, 169, 194
rotation angle 194
RotationInterpolator 194
Rotator.java 618
rounded rectangle 143
rows to be retrieved 453
RowSet command string 520
RowSet implementations 520
RowSet interface 519, 520
RowSetEvent class 520
RowSetListener interface 520
RPC (remote procedure call) security manager 791
RPC (Remote Procedure Calls) 1596
RSA 392, 414, 423
rtexprvalue element 646
Rule of Entity Integrity 451
Rule of Referential Integrity 452
Running tab in Forte 336
runtime package 430
RuntimeException class 370
S
Salary.com 1753
salt 395, 404
sample 147, 151
SampleModel 147
sandbox 10
sandbox security model 10
SAX (Simple API for XML) 1616
scalability 160, 1321
scale 136, 163, 212
scaling 168
scene 161, 162, 167, 169
scene graph 161, 167, 197
SceneGraphObject 162
Schedule interface 1392
scheduled task 1370
scheduleTask method of interface SchedulingService 1392
scheduling bounds 180
scheduling service 1368, 1369, 1380, 1391, 1392
<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchedulingService.ALL constant</td>
<td>1392</td>
</tr>
<tr>
<td>SchedulingService.NONE constant</td>
<td>1392</td>
</tr>
<tr>
<td>SchedulingService.ONE constant</td>
<td>1392</td>
</tr>
<tr>
<td>schema</td>
<td>498</td>
</tr>
<tr>
<td>scope attribute of <code>&lt;jsp:use-Bean&gt;</code> action</td>
<td>618</td>
</tr>
<tr>
<td>search</td>
<td>method of interface LogService 1389</td>
</tr>
<tr>
<td>search time</td>
<td>1551</td>
</tr>
<tr>
<td>SecretKey class</td>
<td>404</td>
</tr>
<tr>
<td>SecretKeyFactory class</td>
<td>404</td>
</tr>
<tr>
<td>Secure Sockets Layer (SSL) 10</td>
<td>secure access to a Web site 532</td>
</tr>
<tr>
<td>secure protocol 568</td>
<td>Secure Sockets Layer (SSL) 10 secure sockets layer (SSL) 423, 424</td>
</tr>
<tr>
<td>secure transaction 389</td>
<td>securing communication 389</td>
</tr>
<tr>
<td>security 10, 387, 1572</td>
<td>security attack 387</td>
</tr>
<tr>
<td>security exception 430</td>
<td>security manager 410, 412, 430</td>
</tr>
<tr>
<td>Security Service 1517, 1526</td>
<td>SecurityException class 369</td>
</tr>
<tr>
<td>ServiceControlled</td>
<td>interface 1243</td>
</tr>
<tr>
<td>ServiceDiscoveryEvent</td>
<td>class 1570</td>
</tr>
<tr>
<td>ServiceDiscoveryManager class</td>
<td>1252, 1571, 1572, 1577</td>
</tr>
<tr>
<td>ServiceDiscoveryListener class</td>
<td>1578</td>
</tr>
<tr>
<td>ServiceFinder class</td>
<td>1378, 1380, 1389</td>
</tr>
<tr>
<td>ServiceIDListener interface</td>
<td>1249</td>
</tr>
<tr>
<td>ServiceInfo class</td>
<td>1395</td>
</tr>
<tr>
<td>ServiceItem class</td>
<td>1220, 1228, 1570</td>
</tr>
<tr>
<td>ServiceRegistration interface</td>
<td>1223</td>
</tr>
<tr>
<td>ServiceRegistrar interface</td>
<td>1204, 1209, 1223, 1236, 1262</td>
</tr>
<tr>
<td>serviceRemoved method</td>
<td>method of interface ServiceDiscoveryListener 1570</td>
</tr>
<tr>
<td>services provided by a component</td>
<td>9, 322</td>
</tr>
<tr>
<td>ServiceTemplate class</td>
<td>1228, 1262, 1287</td>
</tr>
<tr>
<td>servant 11, 13, 532, 534, 543, 553, 661, 717, 761, 995</td>
<td>servlet alias 662</td>
</tr>
<tr>
<td>servlet-class element</td>
<td>546, 559, 584</td>
</tr>
<tr>
<td>servlet container 534, 535</td>
<td>servlet engine 534</td>
</tr>
<tr>
<td>servlet element 546, 559, 568, 577, 584</td>
<td>servlet mapping 546</td>
</tr>
<tr>
<td>servlet-mapping element</td>
<td>546, 547, 559, 568, 577, 584</td>
</tr>
<tr>
<td>servlet-name element</td>
<td>559, 584</td>
</tr>
<tr>
<td>servlet-name element</td>
<td>servlet-name-mapping element 546, 547, 568, 577, 584</td>
</tr>
<tr>
<td>ServiceConfig interface</td>
<td>1011</td>
</tr>
<tr>
<td>ServletConfig interface</td>
<td>535, 581</td>
</tr>
</tbody>
</table>
ShoppingCartHome.java 1084
SHORT 606
SHORT_DESCRIPTION constant of interface Action 44, 45
showInputDialog method of JOptionPane 110
shutdown Cloudscape server 696
shutdown J2EE server 696
sibling node 116
signature 1707, 1714
signedBy 410
significant whitespace 1617
Simple API for XML (SAX) 1616
simple custom tag 639
Simple Network Management Protocol 1320, 1321, 1338
SimpleBeanInfo class 369
simpleContact.xml 1671
SimpleService.java 1597
SimpleUniverse 167, 180, 197
single-quote character 455
Single sign-on 417
Single-Threaded Execution design pattern 25
Singleton design pattern 275, 772
SixFigureJobs 1752
skeleton 1439, 1452, 1453, 1454, 1458, 1463, 1480, 1481, 1497, 1498, 1499, 1502, 1503, 1504, 1510, 1515
singleton class 807
SkillsVillage.com 1746
SKIP_BODY constant 642, 651
SliderFieldPanel.java 348
SliderFieldPanelBeanInfo.java 364
SliderFieldPanelCustomizer.java 376
SMALL_ICON constant of interface Action 43, 45
Smalltalk-80 86
smart agents 1321
smart Web service 1595, 1609
snapshot entry 1281
snapshot method of interface JavaSpace 1265, 1281
SnapshotUsage.java 1281
SNMP 1366
SNMP (Simple Network Management Protocol) 1320, 1321, 1338
SOAP 16
SOAP (Simple Object Access Protocol) 16, 1595, 1602
SOAP message 1596
SOAP package 1601
SOAP request 1598
SOAP-response message 1596
SOAPAction header 1601
SOAPException class 1601
socket 532
socket-based communication 532
soft-button 773
Solaris SPARC 1322
source code Forte generates 327
Source Editor window in Forte 326, 333, 340
source of an event 333
spatial coordinate 168
spatial transformation 180
Specification Lead (Spec Lead) 1703, 1704
specify attributes of a custom tag 643
Sphere 195, 196
SplashScreen.java 317
SQL (Structured Query Language) 446, 453, 460, 479, 480, 481, 1088
SQL keyword 452
SQL query 1088, 1126
SQL script 463
SQL statement 447, 482, 498, 515, 1155
SQLException class 466, 468, 469, 474, 481, 516, 891, 916
square bracket ([ ] 1629
square brackets in a query 453
SSLava 424
standalone 1635
standard action 604
standard MBean 1323
standard service 1369
start method from RMICone
ctorServer 1339
start persistent JavaSpaces service 1261
start the RMI registry 807
startAnimation 322
startApp method of class MIDlet 761, 774
Starting Directory 978
StartService GUI 1199
StartService tool 1202
stateful session EJB 847, 849, 850, 1075
stateful-session-persist-ent-store-dir 1176
stateless protocol 560
stateless session EJB 847, 849, 869, 968
Statement interface 465, 468, 469, 475, 480, 515, 516, 517, 739
static agents 1320
Static Invocation Interface (SII) 1456, 1502
static services 1366, 1369, 1380, 1381
static XHTML document 662
station 1369, 1379, 1395, 1424, 1429
StationAddress class 1395, 1424
sticky load balancing 1164
StoppableChatServer.java 811
stored procedure 515
Strategy design pattern 24
StreamMessage class 939
StreamResult class 680
StreamSource class 680, 741, 1011, 1679
String 180
stringified IOR 1481, 1504
StringItem class 780
StringTokenizer class 776
Stroke 138, 139
stroke 139
struct CORBA keyword 1442, 1459, 1461, 1462, 1463, 1480, 1481, 1492, 1493, 1494, 1495, 1503, 1504
structural design patterns 21
Structured Query Language (SQL) 10, 446, 447, 452
stub 1439, 1441, 1452, 1453, 1454, 1456, 1463, 1465, 1474, 1477, 1481, 1488, 1497, 1498, 1499, 1502, 1503, 1504, 1510, 1515
stub class 807
style attribute 643
style sheet 665
styles.css 663, 666
subclass 23
subject 88
subscribeObserver method of interface EventService 1382
subscriber 963
subscribeResponsible method of interface 1381
subscription 951
substitution cipher 388
Subsystem 1370
Sum (SQL) 582
Sun Microsystems, Inc. 1764
Sun Microsystems, Inc. Java site 585
Sun MIDP-device emulator 720, 774
sun-util.jar 1198, 1269, 1272, 1273, 1280
sun.jdbc.odbc 467
supplier 1521
Support class 371
Supports transaction type 891
Survey.html 582
SurveyServlet.java 578
Swing 8, 30, 178
Swing component 87
Swing Forms option 325
Swing tab in the Component Palette 331
Switch 180, 196, 197
Sybase 446
Sybase, Inc. 1764
symbol 18, 1763
symmetric cryptography 389, 390
symmetric key algorithm 393
synchronization 1145
synchronized 25
SYSTEM 1629, 1629, 1629, 1629, 1644, 1645
System.loadLibrary method 1706
systemclock.idl 1443
SystemClockClient.java 1449, 1511
SystemClockImpl.java 1445
SystemColor 142
SystemException class 886
T
tab 77
table 446, 479
table column 447
table in which record will be updated 461, 462
table-name 1178
TableModel interface 111, 470
TableModelEvent class 476
tableName .fieldName 459
tag element 642
tag extension mechanism 595
tag handler 639, 640, 641, 645, 647
Tag interface 640, 641, 642
tag library 594, 595, 634, 638
tag library descriptor 639, 642, 646, 652
tagclass element 642, 646, 652
TagExtraInfo class 651
tagging interface 792
taglib directive 634, 639, 640, 646
taglib directive attributes 639
tagPrefix attribute of taglib directive 639
TagSupport class 638, 640, 641
take method of interface JavaSpace 1264, 1273
take operation 1259
takeIfExists method of interface JavaSpace 1264, 1273
TakeOperation.java 1273
TallyDisplay.java 975
TallyPanel.java 950, 977
target of an event 333, 384
Target Server 868
TCK 1704
TCP/IP 423, 424, 542, 1440
Technology Compatibility Kit (TCK) 1703
teliclass element of tag library descriptor 652
telephone system 1550
template 17
template 1682
Template Chooser dialog 325
template matching 1260, 1269
Template Method design pattern 272
Text interface 1658
text/html MIME type 539, 541
Text3D 180, 195
TextBox class 772
TextInputPanel class 272
TextMessage 941, 949
TextMessage class 939, 940
Texture 161, 162, 196
texture 162, 196, 212
texturing 180
texture mapping 136, 161, 169, 175
TextureLoader 168
TexturePaint 138, 143
Thawte 409
The Diversity Directory 1745
The National Business and Disability Council (NBDC) 1746
thin client 10, 533
third tier 578
thomas.loc.gov/cgi-bin/bdquery/ z7d106:hr.01714: 407
thomas.loc.gov/cgi-bin/bdquery/ z7d106:s.00761: 407
Thread class 319, 1575
three-tier architecture 717, 992, 1366
three-tier distributed application 578
three-tier architecture 15
Throwable class 1389, 1390
ThrowNew function of C table
JNIEnv 1724
Ticket class 1392
Ticket-Granting Ticket (TGT) 417
TIE (Object Adapter in delegation model) 1516
tier 992
Time To Live (TTL) 1576, 1582, 1583
Timer class 339
timestamping 406
TimeZone class 1391
TintImage.cpp 1724
Tip Test 717, 773, 776, 778, 780, 785
TipTestMidlet 761
TipTestMidlet Tip Test answer screen. 780
TipTestMidlet.java 762
TipTestServlet.java 726
title attribute of <jsp:plugin> action 614
titles table of books database 448, 450
Tomcat server 533, 539, 542, 559
Tomcat shut down 544
Tomcat start 544
tomcat.bat 544
tomcat.sh 544
TOMCAT_HOME environment variable 543
Tomcat documentation home page 544
Tomcat 783
Tomcat documentation home page 544
Tokens 618, 619
Token connection 1638
token 417
tokenized attribute 1638
trace request 537
trans-timeout-seconds 1176
transaction-descriptor 1175
Transaction 1287, 1299
TransactionManager 1287
TransactionManager-Finder.java 1287
Transferable 57, 61, 276, 283, 284
TransferableShape.java 285
transform 139, 194
toolbar 33
Tools menu Add to JAR item 343
Tools menu Install New JavaBean... 323
tooltip 72
top tier 993
topic 938, 951, 957, 963
Topic 957
TopicConnection 957
TopicConnectionFactory 957
TopicPublisher 957
TopicSession 957
TopicSubscriber 957
trans-timeout-seconds 1176
transaction-descriptor 1175
Transaction 1266, 1269
transaction management 1153, 1163
Transaction Management dialog 866, 982
transaction processing 467, 480, 482
transaction service 1368, 1369
Transaction Type 1084, 1147, 1153
Transaction.Created 1289
TransactionException class 1266, 1270
TransactionFactory class 1289
TransactionManager interface 1287, 1299
TransactionManager-Finder.java 1287
Transferable interface 57, 61, 276, 283, 284
TransferableShape.java 285
transform 139, 194
toolbar 33
Tools menu Add to JAR item 343
Tools menu Install New JavaBean... 323
tooltip 72
top tier 993
topic 938, 951, 957, 963
Topic 957
TopicConnection 957
TopicConnectionFactory 957
TopicPublisher 957
TopicSession 957
TopicSubscriber 957
trans-timeout-seconds 1176
transaction-descriptor 1175
Transaction 1266, 1269
transaction management 1153, 1163
Transaction Management dialog 866, 982
transaction processing 467, 480, 482
transaction service 1368, 1369
Transaction Type 1084, 1147, 1153
Transaction.Created class 1289
TransactionException class 1266, 1270
TransactionFactory class 1289
TransactionManager interface 1287, 1299
TransactionManager-Finder.java 1287
Transferable interface 57, 61, 276, 283, 284
TransferableShape.java 285
transform 139, 194
tools attribute of <jsp:plugin> action 613
tools attribute of <jsp:useBean> action 618
type drop-down list 1153
type-storage 1175
type-version 1175
TYPE_FORWARD_ONLY 474
TYPE_INT_RGB 149
TYPE_SCROLL_INSENSITIVE 474
TYPE_SCROLL_SENSITIVE 475
type-identifier 1175
types class 468

U
U+yyyy (Unicode notational convention) 1765, 1771
unambiguous (Unicode design basis) 1764, 1771
underscore (_) SQL wildcard character 455, 456
unicast discovery 1204, 1262, 1552
unicast discovery protocol 14
UnicastDiscovery.java 1204, 1232
UnicastDiscoveryUtility.java 1232
UnicastRemoteObject class 796, 797, 809, 1555, 1561
UnicastSeminarInfoClient.java 1223
Unicode 62, 1708
Unicode Consortium 18, 1616, 1764, 1771, 1772
Unicode escape sequence 68
Unicode Standard 18, 1763, 1771, 1772
Unicode Standard design basis 1764
Unicode Transformation Format (UTF) 1771
Unified Modeling Language (UML) 1543
uniform (Unicode design basis) 1764, 1771
Uniform Resource Identifier (URI) 1596, 1621
unique ID 14, 1118
Universal Description, Discovery and Integration (UDDI) 1595
universal (Unicode design basis) 1771
universal (Unicode design principle) 1764
Universal Resource Locator (URL) 1621
UnknownHostException class 1576
unmodifiableCollection method of class Collections 244
UnsatisfiedLinkError class 1706, 1709
UnsupportedEncodingException class 404
UnsupportedFlavorException class 284
UnusableEntryException class 1270
updatable ResultSet 518
UPDATE 453, 461, 498
update method of interface Observer 89, 93
UPDATE statement 923
UpdateCartServlet.java 1038
UpdateInputWindow.java 1285
UpdateOperation.java 1289
updateRow method of interface ResultSet 518
URI (Uniform Resource Identifier) 534, 1596
uri attribute of taglib directive 639, 640
URI path of a resource 605
URISerializer interface 1011, 1028
URL (Uniform Resource Locator) 534
URL (uniform resource locator) 1621
URL (Universal Resource Locator) 1621
URL class 33, 1011
URL format 534
URL pattern 547
url-pattern element 546, 547, 559, 568, 577, 584
US constant of class Locale 68
usage.xsl 1684
usage1.xsl 1691
usage2.xsl 1690
User-Agent header 722, 739, 775, 786
User-Agent header 1018, 1019
user interface 26, 577, 993
UTF (Unicode Transformation Format) 1771
UTF-8 1615, 1708, 1764, 1771
UTF-16 1615, 1764, 1771
UTF-32 1764, 1771
valid XML document 1628
validating parser 1628
value attribute of <jsp:param> action 610
value attribute of <jsp:set-Property> action 622
valueForPathChanged method of interface TreeModel 128
VALUES 460
valuetype CORBA keyword 1442, 1443, 1495, 1496, 1497, 1504
variable element 1695
variables.xsl 1695
Vault.com 1742
Vector class 111, 1309
verbose output 343
VeriSign 407, 408
version 1613
vertical CORBAfacilities 1457
VERTICAL_SPLIT constant of class JSplitPane 48
video 532
view 8, 86, 87, 88, 91
view (in MVC architecture) 25, 26
viewCart.jsp 663, 664, 686
ViewCartServlet.java 1026
viewdbg tool 1409
viewdbg tool 1409
viewing distance 167, 180
ViewingPlatform 167
viewlog tool 1391, 1427, 1428
ViewOrderHistoryServlet.java 1061
ViewOrderServlet.java 1065
ViewPlatform 162, 197
viewport 162
virtual machine 342, 758
VirtualUniverse 161
visualization 160
Vlissides, John 20
VoteCollector.java 945
VoteCollectorEJB.java 972
VoteListener.java 949
Voter.java 941
vspace attribute of <jsp:plugin> action 614

W
wait method of class Object 1399
wake-up condition 197
WakeupCondition 197
WakeupCriterion 197
WakeupOnCollisionEntry 197
WAP 719, 720, 746, 785, 995
WAPTipAnswer.xsl 749
WAPTipQuestion.xsl 746
WAR (Web application archive) 545, 1159
.war file extension 545
warning method (ErrorHandler) 1664
WeatherBean.java 799
WeatherCellRenderer.java 805
WeatherDisplay.java 965
WeatherItem.java 805
WeatherListener.java 964
WeatherListModel.java 803
WeatherPublisher.java 953
WeatherService.java 793, 1602
WeatherServiceClient.java 801, 1605
WeatherServiceImpl.java 794
WeatherSubscriber.java 958
Web application 545
Web application archive (WAR) 545
Web application deployment descriptor 545
Web application deployment tool 546
Web component 694, 698
web connector 1164
Web context 694, 706
Web form 536, 541
Web server 532, 543, 595, 992, 1199
Web servers that support servlets 543
Web service 1595, 1596, 1602
Web service 16
Web services 5
web.mit.edu/network/ppg.html 392
web.xml 545, 546, 548, 726, 739, 786
web-app element 546, 547
webapps 545
WebBrowser.java 38, 67
WebBrowserPane.java 31
WebHire 1743
WEB-INF 545, 548
WEB-INF/classes 545
WEB-INF/lib 545
WebLogic 991, 1163, 1165, 1166, 1168, 1176, 1178, 1180, 1183
WebLogic_CMP_RDBMS 1175
weblogic-cmp-rdbms-jar-address.xml 1178
weblogic-cmp-rdbms-jar-Customer.xml 1180
weblogic-cmp-rdbms-jar-order.xml 1184
weblogic-ejb-jar.xml 1168
weblogic-enterprise-bean 1168
weblogic-rdbms-bean 1178
weblogic.xml 1190
WebRowSet class 520
WebServer panel 1200
WebSphere 991, 1163, 1165, 1166, 1168, 1176, 1178, 1180, 1183
WelcomeTagHandler.java 641
well-known port number 543
WHERE clause 453, 454, 455, 456, 458, 459, 461, 462, 498, 1156
whitespace 1616
whitespace character 1616
Whitespace characters in an XML document 1616
whitespace.xml 1616, 1646
width attribute of <jsp:plugin> action 614
wild card 1395
Windows 2000 1322
Windows NT 1322
wireless application development 11
wireless application protocol (WAP) 1750
dirent Internet 11
Wireless Markup Language (WML) 993
wireless office 11
wireless phones 1589
WirelessResumes.com 1750
wizard-style interface 1149
WML (Wireless Markup Language) 720, 724, 748, 749, 783, 991, 993
WML browser 992
workflow 1428
WorkingSolo.com 1751
workstation logon script 417
World Wide Web 531
World Wide Web browser 532
World Wide Web server 532
WHERE clause 453, 454, 455, 456, 458, 459, 461, 462, 498, 1156
Xalan parser 1656
XDrive™ 1748
Xerces parser 1656
Index

XHTML 532, 534, 539, 541, 595, 597, 719, 723, 741, 785, 991, 993, 995
XHTML document 683, 693, 694, 698
XHTML table 686
XHTMLTipAnswer.xsl 744
XHTMLTipQuestion.xsl 741
XML 11, 13, 16, 222, 533, 534, 595, 661, 739, 741, 748, 749, 756, 993, 1010, 1101
XML comment (<!---->) 1613
XML declaration 1613
XML deployment descriptor 696
XML descriptor 1153
XML document 1589
XML Document Object Model 1654
XML element 1093, 1113, 1127, 1614
XML end tag 1613
XML namespace 1620
xml namespace 1620
XML parser 1615
XML processor 1615
XML start tag 1613
XML tag 1613
XML4J (XML Parser for Java) 1656
XMLGenerator.java 1093
xmlns attribute 1621
XMLServlet.java 1011
XML-SOAP Admin tool 1597
XSL (Extensible Stylesheet Language) 1677
XSL file 694
XSL transformation 680, 991, 995, 1010
XSL Transformation Language (XSLT) 1677
XSL Transformer 995
xsl:stylesheet element 1679
xsl:value-of element 1680
XSLT 741, 748, 753, 756, 993
XSLT (XSL Transformation Language) 1677, 1680
XSLT processor 1677
XSLT stylesheet 13
XSLT template 1686

Z
ZoomDialog.java 305
ZoomDrawingView.java 258

Y
Yahoo! 1744