Contents
Contents

5  Control Structures: Part 2 197
  5.1 Introduction 198
  5.2 Essentials of Counter-Controlled Repetition 198
  5.3 The for Repetition Structure 201
  5.4 Examples Using the for Structure 205
  5.5 The switch Multiple-Selection Structure 210
  5.6 The do/while Repetition Structure 215
  5.7 Statements break and continue 218
  5.8 Labeled break and continue Statements 220
  5.9 Logical Operators 222
  5.10 Structured Programming Summary 229
  5.11 (Optional Case Study) Thinking About Objects: Identifying Objects’ States and Activities 234

6  Methods 246
  6.1 Introduction 247
  6.2 Program Modules in Java 247
  6.3 Math Class Methods 249
  6.4 Methods 249
  6.5 Method Definitions 251
  6.6 Argument Promotion 258
  6.7 Java API Packages 259
  6.8 Random-Number Generation 261
  6.9 Example: A Game of Chance 265
  6.10 Duration of Identifiers 274
  6.11 Scope Rules 275
  6.12 Recursion 278
  6.13 Example Using Recursion: The Fibonacci Series 281
  6.14 Recursion vs. Iteration 286
  6.15 Method Overloading 288
  6.16 Methods of Class JApplet 291
  6.17 (Optional Case Study) Thinking About Objects: Identifying Class Operations 293

7  Arrays 313
  7.1 Introduction 314
  7.2 Arrays 315
  7.3 Declaring and Allocating Arrays 317
  7.4 Examples Using Arrays 317
    7.4.1 Allocating an Array and Initializing Its Elements 318
    7.4.2 Using an Initializer List to Initialize Elements of an Array 319
    7.4.3 Calculating the Value to Store in Each Array Element 320
    7.4.4 Summing the Elements of an Array 322
    7.4.5 Using Histograms to Display Array Data Graphically 323
    7.4.6 Using the Elements of an Array as Counters 324
    7.4.7 Using Arrays to Analyze Survey Results 326

© Copyright 2002 by Prentice Hall. All Rights Reserved.
Contents

9.15 Polymorphism Examples 475
9.16 Case Study: A Payroll System Using Polymorphism 477
9.17 New Classes and Dynamic Binding 485
9.18 Case Study: Inheriting Interface and Implementation 486
9.19 Case Study: Creating and Using Interfaces 494
9.20 Inner Class Definitions 501
9.21 Notes on Inner Class Definitions 512
9.22 Type-Wrapper Classes for Primitive Types 513
9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation 513
9.24 (Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns 520
  9.24.1 Creational Design Patterns 521
  9.24.2 Structural Design Patterns 523
  9.24.3 Behavioral Design Patterns 524
  9.24.4 Conclusion 526
  9.24.5 Internet and World-Wide-Web Resources 526
10 Strings and Characters 536
  10.1 Introduction 537
  10.2 Fundamentals of Characters and Strings 538
  10.3 String Constructors 538
  10.4 String Methods length, charAt and getChars 540
  10.5 Comparing Strings 542
  10.6 String Method hashCode 547
  10.7 Locating Characters and Substrings in Strings 549
  10.8 Extracting Substrings from Strings 551
  10.9 Concatenating Strings 552
  10.10 Miscellaneous String Methods 553
  10.11 Using String Method valueOf 555
  10.12 String Method intern 557
  10.13 StringBuffer Class 559
  10.14 StringBuffer Constructors 560
  10.15 StringBuffer Methods length, capacity, setLength and ensureCapacity 561
  10.16 StringBuffer Methods charAt, setCharAt, getChars and reverse 563
  10.17 StringBuffer append Methods 564
  10.18 StringBuffer Insertion and Deletion Methods 566
  10.19 Character Class Examples 568
  10.20 Class StringTokenizer 576
  10.21 Card Shuffling and Dealing Simulation 579
  10.22 (Optional Case Study) Thinking About Objects: Event Handling 583
11 Graphics and Java2D 601
  11.1 Introduction 602
  11.2 Graphics Contexts and Graphics Objects 604
11.3 Color Control 605
11.4 Font Control 612
11.5 Drawing Lines, Rectangles and Ovals 618
11.6 Drawing Arcs 622
11.7 Drawing Polygons and Polylines 625
11.8 The Java2D API 628
11.9 Java2D Shapes 628
11.10 (Optional Case Study) Thinking About Objects: Designing Interfaces with the UML 635

12 Graphical User Interface Components: Part 1 646
12.1 Introduction 647
12.2 Swing Overview 649
12.3 JLabel 651
12.4 Event-Handling Model 654
12.5 JTextField and JPasswordField 656
12.5.1 How Event Handling Works 660
12.6 JButton 662
12.7 JCheckBox and JRadioButton 665
12.8 JComboBox 671
12.9 JList 673
12.10 Multiple-Selection Lists 676
12.11 Mouse Event Handling 678
12.12 Adapter Classes 683
12.13 Keyboard Event Handling 689
12.14 Layout Managers 692
12.14.1 FlowLayout 693
12.14.2 BorderLayout 696
12.14.3 GridLayout 699
12.15 Panels 701
12.16 (Optional Case Study) Thinking About Objects: Use Cases 703

13 Graphical User Interface Components: Part 2 720
13.1 Introduction 721
13.2 JTextArea 722
13.3 Creating a Customized Subclass of JPanel 725
13.4 Creating a Self-Contained Subclass of JPanel 730
13.5 JSlider 735
13.6 Windows 739
13.7 Designing Programs that Execute as Applets or Applications 741
13.8 Using Menus with Frames 747
13.9 Using JPopupMenu 755
13.10 Pluggable Look-and-Feel 758
13.11 Using JDesktopPane and JInternalFrame 762
13.12 Layout Managers 766
13.13BoxLayout Layout Manager 767
13.14 CardLayout Layout Manager 770

©Copyright 2002 by Prentice Hall. All Rights Reserved.
## Contents

13.15 **GridBagLayout** Layout Manager 774  
13.16 **GridBagConstraints** Constants RELATIVE and REMAINDER 780  
13.17 (Optional Case Study) Thinking About Objects: Model-View-Controller 783  
13.18 (Optional) Discovering Design Patterns: Design Patterns Used in Packages `java.awt` and `javax.swing` 788  
13.18.1 Creational Design Patterns 789  
13.18.2 Structural Design Patterns 789  
13.18.3 Behavioral Design Patterns 792  
13.18.4 Conclusion 795  

14 **Exception Handling** 804  
14.1 Introduction 805  
14.2 When Exception Handling Should Be Used 807  
14.3 Other Error-Handling Techniques 807  
14.4 Basics of Java Exception Handling 808  
14.5 `try` Blocks 809  
14.6 Throwing an Exception 809  
14.7 Catching an Exception 810  
14.8 Exception-Handling Example: Divide by Zero 812  
14.9 Rethrowing an Exception 818  
14.10 `throws` Clause 818  
14.11 Constructors, Finalizers and Exception Handling 824  
14.12 Exceptions and Inheritance 824  
14.13 `finally` Block 825  
14.14 Using `printStackTrace` and `getMessage` 830  

15 **Multithreading** 837  
15.1 Introduction 838  
15.2 Class `Thread`: An Overview of the `Thread` Methods 840  
15.3 Thread States: Life Cycle of a Thread 841  
15.4 Thread Priorities and Thread Scheduling 842  
15.5 Thread Synchronization 848  
15.6 Producer/Consumer Relationship without Thread Synchronization 849  
15.7 Producer/Consumer Relationship with Thread Synchronization 854  
15.8 Producer/Consumer Relationship: The Circular Buffer 860  
15.9 Daemon Threads 869  
15.10 `Runnable` Interface 870  
15.11 Thread Groups 876  
15.12 (Optional Case Study) Thinking About Objects: Multithreading 877  
15.13 (Optional) Discovering Design Patterns: Concurrent Design Patterns 886  

16 **Files and Streams** 894  
16.1 Introduction 895  
16.2 Data Hierarchy 895  
16.3 Files and Streams 897  
16.4 Creating a Sequential-Access File 903  
16.5 Reading Data from a Sequential-Access File 915  

© Copyright 2002 by Prentice Hall. All Rights Reserved.
16.6 Updating Sequential-Access Files 927
16.7 Random-Access Files 928
16.8 Creating a Random-Access File 928
16.9 Writing Data Randomly to a Random-Access File 933
16.10 Reading Data Sequentially from a Random-Access File 939
16.11 Example: A Transaction-Processing Program 944
16.12 Class File 961

17 Networking 978
17.1 Introduction 979
17.2 Manipulating URIs 981
17.3 Reading a File on a Web Server 986
17.4 Establishing a Simple Server Using Stream Sockets 990
17.5 Establishing a Simple Client Using Stream Sockets 991
17.6 Client/Server Interaction with Stream Socket Connections 992
17.7 Connectionless Client/Server Interaction with Datagrams 1003
17.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server 1011
17.9 Security and the Network 1026
17.10 DeitelMessenger Chat Server and Client 1026
17.10.1 DeitelMessengerServer and Supporting Classes 1027
17.10.2 DeitelMessenger Client and Supporting Classes 1036
17.11 (Optional) Discovering Design Patterns: Design Patterns Used in Packages java.io and java.net 1056
17.11.1 Creational Design Patterns 1056
17.11.2 Structural Design Patterns 1057
17.11.3 Architectural Patterns 1058
17.11.4 Conclusion 1060

18 Multimedia: Images, Animation, Audio and Video 1068
18.1 Introduction 1069
18.2 Loading, Displaying and Scaling Images 1070
18.3 Animating a Series of Images 1073
18.4 Customizing LogoAnimator via Applet Parameters 1077
18.5 Image Maps 1081
18.6 Loading and Playing Audio Clips 1084
18.7 Internet and World Wide Web Resources 1087

19 Data Structures 1094
19.1 Introduction 1095
19.2 Self-Referential Classes 1096
19.3 Dynamic Memory Allocation 1096
19.4 Linked Lists 1097
19.5 Stacks 1108
19.6 Queues 1113
19.7 Trees 1116

© Copyright 2002 by Prentice Hall. All Rights Reserved.
## 20 Java Utilities Package and Bit Manipulation 1147

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Introduction</td>
<td>1148</td>
</tr>
<tr>
<td>20.2 Vector Class and Enumeration Interface</td>
<td>1148</td>
</tr>
<tr>
<td>20.3 Stack Class</td>
<td>1156</td>
</tr>
<tr>
<td>20.4 Dictionary Class</td>
<td>1160</td>
</tr>
<tr>
<td>20.5 Hashtable Class</td>
<td>1161</td>
</tr>
<tr>
<td>20.6 Properties Class</td>
<td>1168</td>
</tr>
<tr>
<td>20.7 Random Class</td>
<td>1174</td>
</tr>
<tr>
<td>20.8 Bit Manipulation and the Bitwise Operators</td>
<td>1175</td>
</tr>
<tr>
<td>20.9 BitSet Class</td>
<td>1190</td>
</tr>
</tbody>
</table>

## 21 Collections 1201

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1 Introduction</td>
<td>1202</td>
</tr>
<tr>
<td>21.2 Collections Overview</td>
<td>1203</td>
</tr>
<tr>
<td>21.3 Class Arrays</td>
<td>1203</td>
</tr>
<tr>
<td>21.4 Interface Collection and Class Collections</td>
<td>1208</td>
</tr>
<tr>
<td>21.5 Lists</td>
<td>1208</td>
</tr>
<tr>
<td>21.6 Algorithms</td>
<td>1215</td>
</tr>
<tr>
<td>21.6.1 Algorithm sort</td>
<td>1215</td>
</tr>
<tr>
<td>21.6.2 Algorithm shuffle</td>
<td>1217</td>
</tr>
<tr>
<td>21.6.3 Algorithms reverse, fill, copy, max and min</td>
<td>1219</td>
</tr>
<tr>
<td>21.6.4 Algorithm binarySearch</td>
<td>1221</td>
</tr>
<tr>
<td>21.7 Sets</td>
<td>1223</td>
</tr>
<tr>
<td>21.8 Maps</td>
<td>1226</td>
</tr>
<tr>
<td>21.9 Synchronization Wrappers</td>
<td>1228</td>
</tr>
<tr>
<td>21.10 Unmodifiable Wrappers</td>
<td>1228</td>
</tr>
<tr>
<td>21.11 Abstract Implementations</td>
<td>1229</td>
</tr>
<tr>
<td>21.12 (Optional) Discovering Design Patterns: Design Patterns Used in Package java.util</td>
<td>1229</td>
</tr>
<tr>
<td>21.12.1 Creational Design Patterns</td>
<td>1229</td>
</tr>
<tr>
<td>21.12.2 Behavioral Design Patterns</td>
<td>1230</td>
</tr>
<tr>
<td>21.12.3 Conclusion</td>
<td>1230</td>
</tr>
</tbody>
</table>

## 22 Java Media Framework and Java Sound (on CD) 1236

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1 Introduction</td>
<td>1237</td>
</tr>
<tr>
<td>22.2 Playing Media</td>
<td>1238</td>
</tr>
<tr>
<td>22.3 Formatting and Saving Captured Media</td>
<td>1249</td>
</tr>
<tr>
<td>22.4 RTP Streaming</td>
<td>1263</td>
</tr>
<tr>
<td>22.5 Java Sound</td>
<td>1277</td>
</tr>
<tr>
<td>22.6 Playing Sampled Audio</td>
<td>1278</td>
</tr>
<tr>
<td>22.7 Musical Instrument Digital Interface (MIDI)</td>
<td>1285</td>
</tr>
<tr>
<td>22.7.1 MIDI Playback</td>
<td>1286</td>
</tr>
<tr>
<td>22.7.2 MIDI Recording</td>
<td>1291</td>
</tr>
<tr>
<td>22.7.3 MIDI Synthesis</td>
<td>1295</td>
</tr>
<tr>
<td>22.7.4 Class MidiDemo</td>
<td>1299</td>
</tr>
<tr>
<td>22.8 Internet and World Wide Web Resources</td>
<td>1316</td>
</tr>
</tbody>
</table>
22.9 (Optional Case Study) Thinking About Objects: Animation and Sound in the View 1317

A Java Demos 1346
A.1 Introduction 1346
A.2 The Sites 1346

B Java Resources 1348
B.1 Resources 1348
B.2 Products 1349
B.3 FAQs 1350
B.4 Tutorials 1350
B.5 Magazines 1350
B.6 Java Applets 1350
B.7 Multimedia 1351
B.8 Newsgroups 1351

C Operator Precedence Chart 1353

D ASCII Character Set 1355

E Number Systems (on CD) 1356
E.1 Introduction 1357
E.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers 1360
E.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers 1361
E.4 Converting from Binary, Octal, or Hexadecimal to Decimal 1361
E.5 Converting from Decimal to Binary, Octal, or Hexadecimal 1362
E.6 Negative Binary Numbers: Two’s Complement Notation 1364

F Creating HTML Documentation with javadoc (on CD) 1369
F.1 Introduction 1370
F.2 Documentation Comments 1370
F.3 Documenting Java Source Code 1370
F.4 javadoc 1379
F.5 Files Produced by javadoc 1379

G Elevator Events and Listener Interfaces (on CD) 1384
G.1 Introduction 1384
G.2 Events 1384
G.3 Listeners 1388
G.4 Component Diagrams Revisited 1391

H Elevator Model (on CD) 1393
H.1 Introduction 1393
H.2 Class ElevatorModel 1393
H.3 Classes Location and Floor 1401
H.4 Class Door 1404
H.5 Class Button 1408

©Copyright 2002 by Prentice Hall. All Rights Reserved.
Contents

H.6 Class ElevatorShaft 1409
H.7 Classes Light and Bell 1416
H.8 Class Elevator 1420
H.9 Class Person 1429
H.10 Component Diagrams Revisited 1436
H.11 Conclusion 1436

I Elevator View (on CD) 1438
I.1 Introduction 1438
I.2 Class Objects 1455
I.3 Class Constants 1457
I.4 Class constructor 1458
I.5 Event Handling 1460
   I.5.1 ElevatorMoveEvent types 1461
   I.5.2 PersonMoveEvent types 1461
   I.5.3 DoorEvent types 1462
   I.5.4 ButtonEvent types 1462
   I.5.5 BellEvent types 1463
   I.5.6 LightEvent types 1463
I.6 Component Diagrams Revisited 1463
I.7 Conclusion 1463

J Career Opportunities (on CD) 1465
J.1 Introduction 1466
J.2 Resources for the Job Seeker 1467
J.3 Online Opportunities for Employers 1468
   J.3.1 Posting Jobs Online 1470
   J.3.2 Problems with Recruiting on the Web 1472
   J.3.3 Diversity in the Workplace 1472
J.4 Recruiting Services 1473
   J.4.1 Testing Potential Employees Online 1474
J.5 Career Sites 1475
   J.5.1 Comprehensive Career Sites 1475
   J.5.2 Technical Positions 1476
   J.5.3 Wireless Positions 1477
   J.5.4 Contracting Online 1477
   J.5.5 Executive Positions 1478
   J.5.6 Students and Young Professionals 1479
   J.5.7 Other Online Career Services 1480
J.6 Internet and World Wide Web Resources 1481

K Unicode® (on CD) 1489
K.1 Introduction 1490
K.2 Unicode Transformation Formats 1491
K.3 Characters and Glyphs 1492
K.4 Advantages/Disadvantages of Unicode 1493
K.5 Unicode Consortium’s Web Site 1493

©Copyright 2002 by Prentice Hall. All Rights Reserved.
Contents

XVIII

K.6 Using Unicode 1494
K.7 Character Ranges 1497

Bibliography 1501

Index 1506
Preface

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

Welcome to Java How to Program, Fourth Edition and the exciting world of programming with the Java™ 2 Platform, Standard Edition. This book is by an old guy and a young guy. The old guy (HMD; Massachusetts Institute of Technology 1967) has been programming and/or teaching programming for 40 years. The young guy (PJD; MIT 1991) has been programming and/or teaching programming for 22 years, and is both a Sun Certified Java Programmer and a Sun Certified Java Developer. The old guy programs and teaches from experience; the young guy does so from an inexhaustible reserve of energy. The old guy wants clarity; the young guy wants performance. The old guy seeks elegance and beauty; the young guy wants results. We got together to produce a book we hope you will find informative, challenging and entertaining.

In November 1995, we attended an Internet/World Wide Web conference in Boston to hear about Java. A Sun Microsystems representative spoke on Java in a packed convention ballroom. During that presentation, we saw the future of programming unfold. The first edition of Java How to Program was born at that moment and was published as the world’s first Java computer science textbook.

The world of Java is evolving so rapidly that Java How to Program: Fourth Edition is being published less than five years after the first edition. This creates tremendous challenges and opportunities for us as authors, for our publisher—Prentice Hall, for instructors, for students and for professional people.

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 strategic planning and implementation. Organizations want to integrate the Internet “seamlessly” into their information systems. Java is more appropriate than C++ for this purpose.
New Features in Java How to Program: Fourth Edition

This edition contains many new features and enhancements including:

• **Full-Color Presentation.** The book is now in full color. In the book’s earlier two-color editions, the programs were displayed in black and the screen captures appeared in the second color. Full color enables readers to see sample outputs as they would appear on a color monitor. Also, we now syntax color all the Java code, as many of today’s Java development environments do. Our syntax-coloring conventions are as follows:

  - comments appear in green
  - keywords appear in dark blue
  - constants and literal values appear in light blue
  - class, method and variable names appear in black

• **“Code Washing.”** This is our own term for the process we used to convert all the programs in the book to a more open layout with enhanced commenting. We have grouped program code into small, well-documented pieces. This greatly improves code readability—an especially important goal for us given that this new edition contains more than 25,000 lines of code.

• **Tune-Up.** We performed a substantial tune-up of the book’s contents based on our own notes from extensive teaching in our professional Java seminars. In addition, a distinguished team of reviewers read the third edition book and provided us with their comments and criticisms. There are literally thousands of fine-tuning improvements over the third edition.

• **Thinking About Objects.** This optional 180-page case study introduces object-oriented design (OOD) with the Unified Modeling Language (the UML). Many chapters in this edition end with a “Thinking About Objects” section in which we present a carefully paced introduction to object orientation. Our goal in these sections is to help you develop an object-oriented way of thinking to be able to design and implement more substantial systems. These sections also introduce you to the Unified Modeling Language (UML). The UML is a graphical language that allows people who build systems (e.g., software architects, systems engineers and programmers) to represent their object-oriented designs using a common notation. The “Thinking About Objects” section in Chapter 1 introduces basic concepts and terminology. Chapters 2–13, 15 and 22 (22 is on the CD) and Appendices G, H and I (also on the CD) include optional “Thinking About Objects” sections that present a substantial object-oriented elevator case study that applies the techniques of object-oriented design (OOD). Appendices G, H and I fully implement the case study design in Java code. This case study will help prepare you for the kinds of substantial projects you are likely to encounter in industry. If you are a student and your instructor does not plan to include this case study in your course, you may want to read the case study on your own. We believe it will be well worth your effort to walk through this large and challenging project. The material presented in the case-study sections reinforces the material covered in the corresponding chapters. You will experience a solid introduction to object-oriented design with the UML. Also, you will sharpen your code-reading skills by touring
Appendix Preface XXXVII

a carefully written and well-documented 3,465-line Java program that completely solves the problem presented in the case study.

- **Discovering Design Patterns.** These optional sections introduce popular object-oriented design patterns in use today. Most of the examples provided in this book contain fewer than 150 lines of code. Such small examples normally do not require an extensive design process. However, some programs, such as our optional elevator-simulation case study, are more complex—they can require thousands of lines of code. Larger systems, such as automated teller machines or air-traffic control systems, could contain millions, or even hundreds of 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 present several design patterns in Java, but these design patterns can be implemented in any object-oriented language, such as C++, C# or Visual Basic. We describe several design patterns used by Sun Microsystems in the Java API. We use design patterns in many programs in this book, which we will identify in our “Discovering Design Patterns” sections. These programs provide examples of using design patterns to construct reliable, robust object-oriented software.

- **Chapter 22 (on the CD), Java Media Framework (JMF) and JavaSound.** This chapter introduces to Java’s audio and video capabilities, enhancing our Chapter 18 multimedia coverage. With the Java Media Framework, a Java program can play audio and video media, and capture audio and video media from devices such as microphones and video cameras. The JMF enables Java developers to create streaming media applications, in which a Java program sends live or recorded audio or video feeds across the Internet to other computers, then applications on those other computers play the media as it arrives over the network. The Java-Sound APIs enable programs to manipulate MIDI (Musical Instrument Digital Interface) sounds and captured media (i.e., media from a device such as a microphone). The chapter concludes with a substantial MIDI-processing application that enables users to record MIDI files or select MIDI files to play. Users can create their own MIDI music by interacting with the application’s simulated synthesizer keyboard. The application can synchronize playing the notes in a MIDI file with pressing the keys on the simulated synthesizer keyboard—similar to a player piano. [Note: Chapters 18 and 22 both provide substantial sets of exercises. Each chapter also has a special section containing additional interesting and challenging multimedia projects. These are intended only as suggestions for major projects. Solutions are not provided for these additional exercises in either the Instructor’s Manual or the Java 2 Multimedia Cyber Classroom.]

- **Enhanced TCP/IP-Based Networking.** We include a new capstone example in Chapter 17 that introduces multicasting for sending information to groups of network clients. This Deitel Messenger case study emulates many of today’s popular

---

1. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software.* (Massachusetts: Addison-Wesley, 1995).
instant-messaging applications that enable computer users to communicate with friends, relatives and co-workers over the Internet. This 1130-line, multithreaded, client/server Java program uses most of the techniques presented to this point in the book.

- **Appendix J (on the CD), Career Opportunities.** This detailed appendix introduces career services on the Internet. We explore online career services from the employer and employee’s perspective. We suggest sites on 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 really expanded his search dramatically.

- **Appendix K (on the CD), 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](http://unicode.org)) and shows a Java example that displays “Welcome” in eight different languages!

- **Java 2 Plug-In Moved to Chapter 3, Introduction to Applets.** Students enjoy seeing immediate results as they execute their Java programs. This is difficult if those programs are Java applets that execute in Web browsers. Most of today’s Web browsers (with the exception of Netscape Navigator 6) do not support Java 2 applets directly, so students must test their applet programs with the appletviewer utility. Sun Microsystems provides the Java 2 Plug-in to enable Java 2 applets to execute in a Web browser that does not support Java 2. The discussion of the Java Plug-in walks the student through the steps necessary to execute an applet in today’s Web browsers.

- **Chapter 22 and Appendices E-K on the CD.** There are so many topics covered in this new edition that we could not fit them all in the book! On the CD that accompanies this book, you will find the following chapter and appendices: Chapter 22, Java Media Framework (JMF) and Java Sound; Appendix E, Number Systems; Appendix F, Creating HTML Documentation with javadoc; Appendix G, Elevator Events and Listener Interfaces; Appendix H, Elevator Model; Appendix I, Elevator View; Appendix J, Career Opportunities; and Appendix K, Unicode.

- **Chapters Moved to Advanced Java™ 2 Platform How to Program.** Four chapters from *Java How to Program, Third Edition* have been moved to our new book *Advanced Java 2 Platform How to Program* and greatly enhanced. These chapters are: Java Database Connectivity (JDBC), Servlets, Remote Method Invocation and JavaBeans. *Advanced Java 2 Platform How to Program* covers each of these topics in more depth. We present the Table of Contents of *Advanced Java 2 Platform How to Program* shortly.
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 object-oriented systems with the UML, and the design-patterns movement was taking shape. *Java How to Program* has many audiences, so we designed the book to be customizable. In particular, we included more than 200 pages of optional material that introduces object-oriented design, the UML and design patterns, and presents a substantial case study in object-oriented design and programming. This material is carefully distributed throughout the book to enable instructors to emphasize “industrial-strength” object-oriented design in their courses.

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) that will be immediately useful to them while in the university environment and when they head into a world in which the Internet and the World Wide Web have a massive prominence. 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 courses Deitel & Associates, Inc. teaches. Once our students enter lab, we can’t hold them back. They eagerly experiment and explore portions of the Java class libraries that we haven’t as yet covered in class. They produce applications that go well beyond anything we’ve ever tried in our introductory C and C++ courses. And they tell us about projects they “can’t wait” to try after the course.

Focus of the Book
Our goal was clear—produce a Java textbook for introductory university-level courses in computer programming for students with little or no programming experience, yet offer the depth and the rigorous treatment of theory and practice demanded by traditional, upper-level courses and that satisfies professionals’ needs. To meet these goals, we produced a comprehensive book, because our text patiently teaches the basics of computer programming and of the Java language (i.e., data types, control structures, methods, arrays, recursion and other “traditional” programming topics); presents key programming paradigms, including object-based programming, object-oriented programming, event-driven programming and concurrent programming; and provides an extensive treatment of the Java class libraries.

Evolution of Java How to Program
*Java How to Program* (first edition) was the world’s first university computer science textbook on Java. We wrote it fresh on the heels of *C How to Program, Second Edition* and *C++ How to Program*. Hundreds of thousands of university students and professional peo-
people worldwide have learned C, C++ and Java from these texts. Upon publication in August, 2001 *Java How to Program, Fourth Edition* will be used in hundreds of universities and thousands of 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 Program*. We carefully monitored the effectiveness of these courses and tuned the material accordingly.

**Conceptualization of Java**

We believe in Java. Its conceptualization (and public release in 1995) by Sun Microsystems, the creators of Java, was brilliant. Sun based the new language on two of the world’s most widely used implementation languages, C and C++. This immediately gave Java a huge pool of highly skilled programmers who were implementing most of the world’s new operating systems, communications systems, database systems, personal computer applications and systems software. Sun removed the messier, more complex and error-prone C/C++ features (such as pointers, operator overloading and multiple inheritance, among others). They kept the language concise by removing special-purpose features that were used by only small segments of the programming community. They made the language truly portable to be appropriate for implementing Internet-based and World-Wide-Web-based applications, and they built in the features people really need such as strings, graphics, graphical user interface components, exception handling, multithreading, multimedia (audio, images, animation and video), file processing, database processing, Internet and World Wide Web-based client/server networking and distributed computing, and prepackaged data structures. 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 World-Wide-Web pages. Instead of Web pages with only text and static graphics, people’s Web pages could now “come alive” with audios, videos, animations, interactivity—and soon, three-dimensional 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-independent, object-oriented code for conventional, Internet-, Intranet- and Extranet-based applications and applets. Java now has 2.5 million developers worldwide—a stunning accomplishment given that it has only been available publicly for six years. No other programming language has ever acquired such a large developer base so quickly.

**Enabling Multimedia-Based Applications and Communications**

The computer field has never seen anything like the Internet/World Wide Web/Java “explosion” occurring today. People want to communicate. People need to communicate. Sure they have been doing that since the dawn of civilization, but computer communications have been mostly limited to digits, alphabetic characters and special characters. Today, we are in the midst of a multimedia revolution. People want to transmit pictures and they want those pictures to be in color. They want to transmit voices, sounds, audio clips and full-motion color video (and they want nothing less than DVD quality). Eventually, people will in-
sist on three-dimensional, moving-image transmission. Our current flat, two-dimensional
television screens will eventually be replaced with three-dimensional versions that turn our living
rooms into “theaters-in-the-round.” Actors will perform their roles as if we were watching
live theater. Our living rooms will be turned into miniature sports stadiums. Our business
offices will enable video conferencing among colleagues half a world apart as if they were
sitting around one conference table. The possibilities are intriguing and Java is playing a
key role in turning many of them into reality.

Teaching Approach

Java How to Program, Fourth Edition contains a rich collection of examples, exercises,
and projects drawn from many fields to provide the student with a chance to solve interest-
ing real-world problems. The book concentrates on the principles of good software engi-
neering and stresses program clarity. 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 the focus of the way we teach and
write about programming, and the focus of each of our multimedia Cyber Classrooms and
Web-based training courses as well. Each new concept is presented in the context of a com-
plete, working Java program (application or applet) immediately followed by one or more
screen captures showing 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 pro-
grams (25,000+ lines of code) is much like entering and running them on a computer.

Java and Swing from Chapter Two!

Java How to Program, Fourth Edition “jumps right in” with object-oriented programming,
applications and the Swing-style GUI components from Chapter 2! People tell us this is a
“gutsy” move, but Java students really want to “cut to the chase.” There is great stuff to be
done in Java so let’s get right to it! Java is not trivial by any means, but it’s fun to program
with and students can see immediate results. Students can get graphical, animated, multi-
media-based, audio-intensive, multithreaded, database-intensive, network-based programs
running quickly through Java’s extensive class libraries of “reusable components.” They
can implement impressive projects. They are typically more creative and productive in a
one- or two-semester course than in C and C++ introductory courses.

World Wide Web Access

All of the code for Java How to Program is on the CD that accompanies this book and is
Please run each program as you read the text. Make changes to the code examples and see
what happens. See how the Java compiler “complains” when you make various kinds of
erors. Immediately see the effects of making changes to the code. It’s a great way to learn
programming by doing programming. [This is copyrighted material. Feel free to use it as
you study Java, but you may not republish any portion of it without explicit permission
from the authors and Prentice Hall.]
Objectives
Each chapter begins with a statement of objectives. This tells the student what to expect and gives the student an opportunity, after reading the chapter, to determine if he or she 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 students 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 student approach the material in top-down fashion. This, too, helps students anticipate what is to come and set a comfortable and effective learning pace.

25,576 Lines of Code in 197 Example Programs (with Program Outputs)
We present Java features in the context of complete, working Java programs. The programs range from just a few lines of code to substantial examples with several hundred lines of code (and 3,465 lines of code for the optional object-oriented elevator simulator example). Students should use the program code from the CD that accompanies the book or download the code from our Web site (www.deitel.com) and run each program while studying that program in the text.

545 Illustrations/Figures
An abundance of charts, line drawings and program outputs is included. The discussion of control structures, for example, features carefully drawn flowcharts. [Note: We do not teach flowcharting as a program development tool, but we do use a brief, flowchart-oriented presentation to specify the precise operation of each of Java’s control structures.]

605 Programming Tips
We have included programming tips to help students focus on important aspects of program development. We highlight hundreds of these 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 a combined six 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.

97 Good Programming Practices
When we teach introductory courses, we state that the “buzzword” of each course is “clarity,” and we highlight as Good Programming Practices techniques for writing programs that are clearer, more understandable, more debuggable, and more maintainable.

199 Common Programming Errors
Students learning a language tend to make certain errors frequently. Focusing on these Common Programming Errors helps students avoid making the same errors and shortens lines outside instructors’ offices during office hours!
### 46 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.

### 67 Performance Tips

In our experience, teaching students to write clear and understandable programs is by far the most important goal for a first programming course. But students want to write the programs that run the fastest, use the least memory, require the smallest number of keystrokes, or dazzle in other nifty ways. Students really care about performance. They want to know what they can do to “turbo charge” their programs. So we have included 67 Performance Tips that highlight opportunities for improving program performance—making programs run faster or minimizing the amount of memory that they occupy.

### 24 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 students write portable code and to provide insights on how Java achieves its high degree of portability. We had many more portability tips in our books, C How to Program and C++ How to Program. We needed fewer Portability Tips in Java How to Program because Java is designed to be portable top-to-bottom (for the most part)—much less effort is required on the Java programmer’s part to achieve portability than with C or C++.

### 134 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. Much of what the student learns here will be useful in upper-level courses and in industry as the student begins to work with large, complex real-world systems.

### 38 Look-and-Feel Observations

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

**Summary (983 Summary bullets)**

Each chapter ends with additional pedagogical devices. We present a thorough, bullet-list-style summary of the chapter. On average, there are 42 summary bullets per chapter. This helps the students review and reinforce key concepts.

**Terminology (2171 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 95 terms per chapter.

**397 Self-Review Exercises and Answers (Count Includes Separate Parts)**

Extensive self-review exercises and answers are included for self-study. This gives the student a chance to build confidence with the material and prepare for the regular exercises. Students should be encouraged to do all the self-review exercises and check their answers.
779 Exercises (Count Includes Separate Parts)
Each chapter concludes with a set of exercises including simple recall of important terminology and concepts; writing individual Java statements; writing small portions of Java methods and classes; writing complete Java methods, classes, applications and applets; and writing major term projects. The large number of exercises across a wide variety of areas 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, short quizzes and major 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 solutions manual only from their regular Prentice Hall representatives. We regret that we cannot provide the solutions to professionals.] Solutions to approximately half of the exercises are included on the Java Multimedia Cyber Classroom, Fourth Edition CD, which also is part of The Complete Java 2 Training Course. For ordering instructions, please see the last few pages of this book or visit www.deitel.com.

Approximately 5300 Index Entries (with approximately 9500 Page References)
We have included an extensive index at the back of the book. This helps the student find any term or concept by keyword. The index is useful to people reading the book for the first time and is especially useful to practicing programmers 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). Students can use the index with the Terminology sections to be sure they have covered the key material of each chapter.

“Double Indexing” of Java Live-Code™ Examples and Exercises
Java How to Program has 197 live-code™ examples and 1176 exercises (including parts). Many of the exercises are challenging problems or projects requiring 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 LoadAudioAndPlay.java and indexed it both alphabetically (in this case under “L”) and as a subindex item under “Examples.” This makes it easier to find examples using particular features. The more substantial exercises, such as “Maze Generator and Walker,” are indexed both alphabetically (in this case under “M”) and as subindex items under “Exercises.”

Bibliography
An extensive bibliography of books, articles and Sun Microsystems Java 2 documentation is included to encourage further reading.

Software Included with Java How to Program, Fourth Edition
There are a number of for-sale Java products available. However, you do not need them to get started with Java. We wrote Java How to Program, Fourth Edition using only the Java 2 Software Development Kit (J2SDK). For your convenience, Sun’s J2SDK version 1.3.1 is included on the CD that accompanies this book. The J2SDK also can be downloaded from the Sun Microsystems Java Web site java.sun.com. With Sun’s cooperation, we
also were able to include on the CD a powerful Java integrated development environment (IDE)—Sun Microsystems’s Forté for Java Community Edition.

Forté for Java Community Edition 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 Forté for Java Community Edition. If you have any questions about using this software, please read the introductory Forté 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 an HTML Web page with links to the Deitel & Associates, Inc. Web site, the Prentice Hall Web site and the many Web sites listed in the appendices. 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, the CD contains Chapter 22 and Appendices E–K.

Ancillary Package for Java How to Program, Fourth Edition

Java How to Program, Fourth Edition 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 and a test bank of multiple choice questions (approximately 2 per book section). In addition, we 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 operator precedence chart, 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.]

Java 2 Multimedia Cyber Classroom, Fourth Edition (CD and Web-Based Training Versions) and The Complete Java 2 Training Course, Fourth Edition

We have prepared an interactive, CD-based, software version of Java How to Program, Fourth Edition called the Java 2 Multimedia Cyber Classroom, Fourth Edition. It is loaded with features for learning and reference. The Cyber Classroom is wrapped with the textbook at a discount in The Complete Java 2 Training Course, Fourth Edition. If you already have the book and would like to purchase the Java 2 Multimedia Cyber Classroom, Fourth Edition separately, please visit www.informit.com/cyberclassrooms. The ISBN# for the Java 2 Multimedia Cyber Classroom, Fourth Edition is 0-13-064935-x. 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. The 197 live-code™ example Java programs in the textbook truly “come alive” in the Cyber Classroom. If you are viewing a program and want to execute it, you simply click
on 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 Paul Deitel will talk about the program and “walk you through” the code.

The Cyber Classroom also provides navigational aids including extensive hyper-linking. 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 key in a term using the “find” feature and the Cyber Classroom will locate 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 hundreds of solved problems from the textbook that 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 hyper-linking 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.

**Advanced Java™ 2 Platform How to Program**

Our companion book—Advanced Java 2 Platform How to Program—focuses on the Java 2 Platform, Enterprise Edition (J2EE), presents advanced Java 2 Platform Standard Edition features and introduces the Java 2 Platform, Micro Edition (J2ME). This book is intended for developers and upper-level university students in advanced courses who already know Java and want a deeper treatment and understanding of the language. The book features our signature live-code™ approach of complete working programs and contains over 37,000 lines of code. The programs are more substantial than those presented in Java How to Program, Fourth Edition. The book expands the coverage of Java Database Connectivity (JDBC), remote method invocation (RMI), servlets and JavaBeans from Java How to Program, Fourth Edition. The book also covers emerging and more advanced Java technologies of concern to enterprise application developers. The Table of Contents for Advanced Java 2 Platform How to Program is: Chapters—Introduction; Advanced Swing Graphical User Interface Components; Model-View-Controller; Graphics Programming with Java 2D and Java 3D; Case Study: A Java2D Application; JavaBeans Component Model; Security; Java Database Connectivity (JDBC); Servlets; Java Server Pages (JSP); Case Study: Servlet and JSP Bookstore; Java 2 Micro Edition (J2ME) and Wireless Internet; Remote Method Invocation (RMI); Session Enterprise JavaBeans (EJBs) and Distributed Transac-
Appendices—Creating Markup with XML; XML Document Type Definitions; XML Document Object Model (DOM); XSL: Extensible Stylesheet Language Transformations; Downloading and Installing J2EE 1.2.1; Java Community Process (JCP); Java Native Interface (JNI); Career Opportunities; Unicode.

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.

Other 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 Tem Nieto, Sean Santry, Jonathan Gadzik, Kate Steinbuhler, Rashmi Jayaprakash and Laura Treibick.

- Tem Nieto is a graduate of the Massachusetts Institute of Technology. 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 *Java How to Program, Fourth Edition* Tem co-authored Chapters 11, 12, 13 and 21 and the Special Section entitled “Building Your Own Compiler” in Chapter 19.

- Sean Santry, a graduate of Boston College (Computer Science and Philosophy) and co-author of *Advanced Java 2 Platform How to Program*, edited Chapter 22 (Java Media Framework and Java Sound), helped update the programs in Chapter 15 (Multithreading), designed and implemented the Deitel Messenger networking application in Chapter 17 (Networking), helped design the optional case study on OOD/UML, reviewed the optional design patterns case study and reviewed the implementation of the elevator simulation for the OOD/UML case study.

- Jonathan Gadzik, a graduate of the Columbia University School of Engineering and Applied Science (BS in Computer Science) co-authored the optional OOD/UML case study and the optional “Discovering Design Patterns” sections. He also implemented the 3,465-line Java program that completely solves the object-oriented elevator simulation exercise presented in the OOD/UML case study.

- Kate Steinbuhler, a graduate of Boston College with majors in English and Communications, co-authored Appendix J, Career Opportunities, and managed the permissions process. Kate is moving on to law school at the University of Pittsburgh—good luck Kate! Thank you for your contributions to three Deitel publications.
• Rashmi Jayaprakash, a graduate of Boston University with a major in Computer Science, co-authored Appendix K, Unicode.

• Laura Treibick, a graduate of University of Colorado at Boulder with a major in Photography and Multimedia, created the delightful animated bug character for the implementation of the OOD/UML case study.

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

• Susan Warren, a Junior in Computer Science at Brown University, and Eugene Izumo, a Sophomore in Computer Science at Brown University, reviewed the entire Fourth Edition; reviewed and updated Chapter 22, Java Media Framework and Java Sound; and updated Appendix A (Java Demos) and Appendix B (Java Resources). Susan and Eugene also worked on many of the book’s ancillary materials, including the solutions to the exercises, true/false questions for the companion Web site (www.prenhall.com/deitel), true/false questions for the Java 2 Multimedia Cyber Classroom and multiple choice questions for the Instructor’s test bank.

• Vincent He, a Senior in Management and Computer Science at Boston College, co-authored Chapter 22, Java Media Framework and Java Sound—one of the most exciting and fun chapters in the book! We are sure you will enjoy the multimedia extravaganza Vincent created for you.

• Liz Rockett, a Senior in English at Princeton University edited and updated Chapter 22, Java Media Framework and Java Sound.

• Chris Henson, a graduate of Brandeis University (Computer Science and History), reviewed Chapter 22, Java Media Framework and Java Sound.

• Christina Carney, a Senior in Psychology and Business at Framingham State College, researched and updated the bibliography, helped prepare the Preface and performed the URL research for the OOD/UML case study and design patterns.

• Amy Gips, a Sophomore in Marketing and Finance at Boston College, updated and added URLs for applets, graphics, Java 2D and Multimedia in Appendices A and B. Amy also researched quotes for Chapter 22 and helped prepare the Preface.

• Varun Ganapathi, a Sophomore in Computer Science and Electrical Engineering at Cornell University, updated Appendix F, Creating HTML Documentation with javadoc.

• Reshma Khilnani, a Junior in Computer Science and Mathematics at the Massachusetts Institute of Technology, worked with Rashmi on the Unicode Appendix

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

---

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 or Marketing. 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.
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. Camille Trentacoste did a marvelous job as production manager.

The *Java 2 Multimedia Cyber Classroom, Fourth Edition* was developed in parallel with *Java How to Program, Fourth Edition*. 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 *Java 2 Multimedia Cyber Classroom, Fourth Edition* 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.

We sincerely appreciate the efforts of our fourth edition reviewers:

*Java How to Program, Fourth Edition Reviewers*

Dibyendu Baksi (Sun Microsystems)
Tim Boudreau (Sun Microsystems)
Michael Bundschuh (Sun Microsystems)
Gary Ginstling (Sun Microsystems)
Tomas Pavek (Sun Microsystems)
Rama Roberts (Sun Microsystems)
Terry Hull (Sera Nova)
Ralph Johnson (“gang-of-four” co-author of the seminal book, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995)
Cameron Skinner (Embarcadero Technologies; OMG)
Michael Chonoles (Lockheed Martin Adv. Concepts; OMG)
Brian Cook (The Technical Resource Connection; OMG)
Akram Al-Rawi (Zayed University)
Charley Bay (Fronte Range Community College)
Clint Bickmore (Fronte Range Community College)
Ron Braithwaite (Nutriware)
Columbus Brown (IBM)
Larry Brown (co-author of *Core Web Programming*)
Dan Corkum (Trillian Software)
Jonathan Earl (Technical Training and Consulting)
Karl Frank (togethersoft.com)
Charles Fry (thesundancekid.org)
Kyle Gabhart (Objective Solutions)
Felipe Gaucho (Softexport)
Rob Gordon (SuffolkSoft, Inc.)
Michelle Guy (XOR)
Christopher Green (Colorado Springs Technical Consulting Group)
Kevlin Henney (Curbralan Limited)
Ethan Henry (Sitiraka Software)
Faisal Kaleem (Florida International University)
Rob Kelly (SUNY)
Scott Kendall (Consultant, UML author)
Sachin Khana (Freelance Java Programmer)
Michael-Franz Mannion (Java Developer)
Julie McVicar (Oakland Community College)
Matt Mitton (Consultant)
Dan Moore (XOR)
Simon North (Synopsys)
Chetan Patel (Lexisnexis)
Brian Pontarelli (Consultant)
Kendall Scott (Consultant, UML author)
Craig Shofding (CAS Training Corp)
Spencer Roberts (Titus Corporation)
Toby Steel (CertaPay)
Stephen Tockey (Lexisnexis)
Kim Topley (Author of *Core Java Foundation Classes* and *Core Swing: Advanced Programming*, both published by Prentice Hall)
Gustavo Toretti (Java Programmer; Campinas University)
Michael Van Kleeck (Director of Technology, Learning.com)
Dave Wagstaff (Sungard)

*Java How to Program, Third Edition Post-Publication Reviewers*
Jonathan Earl (Technical Training Consultants)
Harry Foxwell (Sun Microsystems)
Terry Hull (Sera Nova)
Ron McCarty (Penn State University Behrend Campus)
Bina Ramamurthy (SUNY Buffalo)
Vadim Tkachenko (Sera Nova)

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*

**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 several dozen 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, Dr. 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.

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, satellite 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 to Computers, the Internet and the Web

Objectives

• To understand basic computer science concepts.
• To become familiar with different types of programming languages.
• To introduce the Java development environment.
• To understand Java’s role in developing distributed client/server applications for the Internet and Web.
• To introduce object-oriented design with the UML and design patterns.
• To preview the remaining chapters of the book.

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

High thoughts must have high language.
Aristophanes

The chief merit of language is clearness.
Galen

My object all sublime
I shall achieve in time.
W. S. Gilbert

He had a wonderful talent for packing thought close, and rendering it portable.
Thomas Babington Macaulay

Egad, I think the interpreter is the hardest to be understood of the two!
Richard Brinsley Sheridan
1.1 Introduction

Welcome to Java! We have worked hard to create what we hope will be an informative, entertaining and challenging learning experience for you. Java is a powerful computer programming language that is fun to use for novices and appropriate for experienced programmers building substantial information systems. Java How to Program: Fourth Edition is designed to be an effective learning tool for each of these audiences.

How can one book appeal to both groups? The answer is that the common core of the book emphasizes achieving program clarity through the proven techniques of structured programming and object-oriented programming. Nonprogrammers will learn programming the right way from the beginning. We have attempted to write in a clear and straightforward manner. The book is abundantly illustrated. Perhaps most importantly, the book presents hundreds of working Java programs and shows the outputs produced when those programs are run on a computer. We teach all Java features in the context of complete working Java programs. We call this the live-code™ approach. These examples are available from three locations—they are on the CD that accompanies this book, they may be...
downloaded from our Web site www.deitel.com and they are available on our interactive CD product, the *Java 2 Multimedia Cyber Classroom: Fourth Edition*. The Cyber Classroom’s features and ordering information appear at the back of this book. The Cyber Classroom also contains answers to approximately half of the solved exercises in this book, including short answers, small programs and many full projects. If you purchased *The Complete Java 2 Training Course: Fourth Edition*, you already have the Cyber Classroom.

The early chapters introduce the fundamentals of computers, computer programming and the Java computer programming language. Novices who have taken our courses tell us that the material in those chapters presents a solid foundation for the deeper treatment of Java in the later chapters. Experienced programmers tend to read the early chapters quickly and find that the treatment of Java in the later chapters is rigorous and challenging.

Many experienced programmers have told us that they appreciate our structured programming treatment. Often, they have been programming in structured languages like C or Pascal, but they were never formally introduced to structured programming, so they are not writing the best possible code in these languages. As they review structured programming in the chapters “Control Structures: Part 1” and “Control Structures: Part 2,” they are able to improve their C and Pascal programming styles as well. So whether you are a novice or an experienced programmer, there is much here to inform, entertain and challenge you.

Most people are familiar with the exciting tasks computers perform. Using this textbook, you will learn how to command computers to perform those tasks. It is *software* (i.e., the instructions you write to command computers to perform *actions* and make *decisions*) that controls computers (often referred to as *hardware*), and Java is one of today’s most popular software-development languages. Java was developed by Sun Microsystems and an implementation of it is available free over the Internet from the Sun Web site java.sun.com/j2se.

This book is based on the *Java 2 Platform, Standard Edition*, which describes the Java language, libraries and tools. Other vendors can implement *Java development kits* based on the *Java 2 Platform*. Sun provides an implementation of the *Java 2 Platform, Standard Edition* called the *Java 2 Software Development Kit, Standard Edition (J2SDK)* that includes the minimum set of tools you need to write software in Java. At the time of this publication, the most recent version was J2SDK 1.3.1. You can download future updates to the J2SDK from the Sun Web site java.sun.com/j2se.

Computer use is increasing in almost every field of endeavor. In an era of steadily rising costs, computing costs have been decreasing dramatically due to rapid developments in both hardware and software technology. Computers that might have filled large rooms and cost millions of dollars two decades ago can now be inscribed on the surfaces of silicon chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on earth—it is an ingredient in common sand. Silicon-chip technology has made computing so economical that hundreds of millions of general-purpose computers are in use worldwide helping people in business, industry, government, and in their personal lives. The number of computers worldwide easily could double in the next few years.

This book will challenge you for several reasons. For many years, students learned C or Pascal as their first programming language. They probably learned the programming methodology called *structured programming*. You will learn both structured programming and the exciting newer methodology, *object-oriented programming*. Why do we teach both? We
believe that object orientation is the key programming methodology of the future. You will
build and work with many objects in this course. However, you will discover that the internal
structure of those objects is built with structured programming techniques. Also, the logic of
manipulating objects is occasionally best expressed with structured programming.

Another reason we present both methodologies is the continuing migration from C-based
systems (built primarily with structured programming techniques) to C++ and Java-based
systems (built primarily with object-oriented programming techniques). There is a huge
amount of so-called “legacy C code” in place, because C has been in use for over three
decades. Once people learn C++ or Java, they find these languages to be more powerful than
C. These people often choose to move their programming projects to C++ or Java. They begin
converting their legacy systems and begin employing the object-oriented programming capa-
bilities of C++ or Java to realize the full benefits of these languages. Often, the choice
between C++ and Java is made based on the simplicity of Java compared to C++.

Java has become the language of choice for implementing Internet-based and Intranet-
based applications and software for devices that communicate over a network. Do not be
surprised when your new stereo and other devices in your home will be networked together
by Java technology! Also, do not be surprised when your wireless devices, like cell phones,
pagers and personal digital assistants (PDAs) communicate over the so-called Wireless
Internet via the kind of Java-based networking protocols that you will learn in this book and
its companion Advanced Java 2 Platform How to Program.

Java is a particularly attractive first programming language. At the JavaOne™ trade
show in June 2001, it was announced that Java is now a required part of the programming
languages curriculum in 56% of US colleges and universities. Also, 87% of US colleges
and universities offer Java courses. Java is attractive to high schools as well. In 2003, the
College Board will standardize on Java for Advanced Placement computer science courses.

Java has evolved rapidly into the large-scale applications arena. Java is no longer a lan-
guage used simply to make World Wide Web pages “come alive.” Java has become the pre-
ferred language for meeting many organizations’ programming needs.

For many years, languages like C and C++ appealed to universities because of their
portability. Introductory courses could be offered in these languages on any hardware/oper-
ating system combination, as long as a C/C++ compiler was available. However, the pro-
gramming world has become more complex and more demanding. Today, users want
applications with graphical user interfaces (GUIs). They want applications that use multi-
media capabilities such as graphics, images, animation, audio and video. They want applica-
tions that can run on the Internet and the World Wide Web and communicate with other
applications. They want applications that can take advantage of the flexibility and perfor-
ance improvements of multithreading (which enables programmers to specify that sev-
eral activities should occur in parallel). They want applications with richer file processing
than is provided by C or C++. They want applications that are not limited to the desktop or
even to some local computer network, but can integrate Internet components and remote
databases as well. They want applications that can be written quickly and correctly in a
manner that takes advantage of prebuilt software components. They want easy access to a
growing universe of reusable software components. Programmers want all these benefits in
a truly portable manner, so that applications will run without modification on a variety of
platforms (i.e., different types of computers running different operating systems). Java
offers all these benefits to the programming community.
Another reason Java is attractive for university courses is that it is fully object oriented. One reason that C++ use has grown so quickly is that it extends C programming into the arena of object orientation. For the huge community of C programmers, this has been a powerful advantage. C++ includes ANSI/ISO C and offers the ability to do object-oriented programming as well. (ANSI is the American National Standards Institute, and ISO is the International Standards Organization.) An enormous amount of C code has been written in industry over the last several decades. C++ is a superset of C, so many organizations find it to be an ideal next step. Programmers can take their C code, compile it (often with nominal changes) in a C++ compiler and continue writing C-like code while mastering the object paradigm. Then, the programmers can gradually migrate portions of the legacy C code into C++ as time permits. New systems can be entirely written in object-oriented C++. Such strategies have been appealing to many organizations. The downside is that, even after adopting this strategy, companies tend to continue producing C-like code for many years. This, of course, means that they do not realize the benefits of object-oriented programming quickly and could produce programs that are confusing and hard to maintain as a result of their hybrid design. Many organizations would prefer to plunge 100% into object-oriented development, but the realities of mountains of legacy code and the temptation to take a C-programming approach often prevent this.

Java is a fully object-oriented language with strong support for proper software engineering techniques. It is difficult to write C-like, so-called procedural programs in Java. You must create and manipulate objects. Error processing is built into the language. Many of the complex details of C and C++ programming that prevent programmers from “looking at the big picture” are not included in Java. For universities, these features are powerfully appealing. Students will learn object-oriented programming from the start. They will simply think in an object-oriented manner.

Here, too, there is a trade-off. Organizations turning to Java for new applications development do not want to convert all their legacy code to Java. So Java allows for so-called native code. This means that existing C and C++ code can be integrated with Java code. Although this may seem a bit awkward (and it certainly can be), it presents a pragmatic solution to a problem most organizations face.

The fact that Java is free for download at the Sun Web site, java.sun.com/j2se, is appealing to universities facing tight budgets and lengthy budget planning cycles. Also, as bug fixes and new versions of Java are developed, these become available immediately over the Internet, so universities can keep their Java software current.

Can Java be taught in a first programming course—the intended audience for this book? We think so. Prior to writing this book, Deitel & Associates, Inc. instructors taught hundreds of Java courses to several thousand people at all levels of expertise, including many nonprogrammers. We found that nonprogrammers become productive faster with Java than with C or C++. They are anxious to experiment with Java’s powerful features for graphics, graphical user interfaces, multimedia, animation, multithreading, networking and the like—and they are successful at building substantial Java programs even in their first courses.

For many years, the Pascal programming language was the preferred vehicle for use in introductory and intermediate programming courses. Many people said that C was too difficult a language for these courses. In 1992, we published the first edition of C How to Program, to encourage universities to try C instead of Pascal in these courses. We used the same pedagogic approach we had used in our university courses for a dozen years, but
wrapped the concepts in C rather than Pascal. We found that students were able to handle C at about the same level as Pascal. However, there was one noticeable difference—students appreciated that they were learning a language (C) likely to be valuable to them in industry. Our industry clients appreciated the availability of C-literate graduates who could work immediately on substantial projects rather than first having to go through costly and time-consuming training programs.

The first edition of *C How to Program* included a 60-page introduction to C++ and object-oriented programming. We saw C++ coming on strong, but we felt it would be at least a few more years before the universities would be ready to teach C++ and object-oriented programming (OOP) in introductory courses.

During 1993, we saw a surge in interest in C++ and OOP among our industry clients, but we still did not sense that the universities were ready to switch to C++ and OOP en masse. So, in January 1994, we published the Second Edition of *C How to Program* with a 300-page section on C++ and OOP. In May 1994, we published the first edition of *C++ How to Program*, a 950-page book devoted to the premise that C++ and OOP were now ready for prime time in introductory university courses for many schools that wanted to be at the leading edge of programming-languages education.

In 1995, we were following the introduction of Java carefully. In November 1995, we attended an Internet conference in Boston. A representative from Sun Microsystems gave a presentation on Java that filled one of the large ballrooms at the Hynes Convention Center. As the presentation proceeded, it became clear to us that Java would play a significant part in the development of interactive, multimedia Web pages. We immediately saw a much greater potential for the language. We saw Java as the proper language for universities to teach first-year programming language students in this modern world of graphics, images, animation, audio, video, database, networking, multithreading and collaborative computing. At the time, we were busy writing the second edition of *C++ How to Program*. We discussed with our publisher, Prentice Hall, our vision of Java making a strong impact in the university curriculum. We all agreed to delay the second edition of *C++ How to Program* a bit so that we could get the first edition of *Java How to Program* (based on Java 1.0.2) to the market in time for fall 1996 courses.

As Java rapidly evolved to Java 1.1, we wrote *Java How to Program: Second Edition* in 1997, less than a year after the first edition reached bookstores. Hundreds of universities and corporate training programs worldwide used the second edition. To keep pace with the enhancements in Java, we published *Java How to Program: Third Edition* in 1999. The third edition was a major overhaul to upgrade the book to the Java 2 Platform.

Java continues to evolve rapidly, so we wrote this fourth edition of *Java How to Program*—our first book to reach a fourth edition—just five years after the first edition was published. This edition is based on the *Java 2 Platform, Standard Edition (J2SE)*. Java has grown so rapidly over the last several years that it now has two other editions. The *Java 2 Platform, Enterprise Edition (J2EE)* is geared toward developing large-scale, distributed networking applications and Web-based applications. The *Java 2 Platform, Micro Edition (J2ME)* is geared toward development of applications for small devices (such as cell phones, pagers and personal digital assistants) and other memory-constrained applications. The number of topics to cover in Java has become far too large for one book. So, in parallel with *Java How to Program, Fourth Edition*, we are publishing *Advanced Java 2 Platform How to Program*, which emphasizes developing applications with J2EE and provides cov-
erage of several high-end topics from the J2SE. In addition, this book also includes sub-
stantial materials on J2ME and wireless-application development.

So, there you have it! You are about to start on a challenging and rewarding path. As
you proceed, please share your thoughts on Java and Java How to Program: Fourth Edition
with us via e-mail at deitel@deitel.com. We will respond promptly.

Prentice Hall maintains www.prenhall.com/deitel—a Web site dedicated to
our Prentice Hall publications, including textbooks, professional books, interactive multi-
media CD-based Cyber Classrooms, Complete Training Courses (boxed products con-
taining both a Cyber Classroom and the corresponding book), Web-based training, e-
whitepapers, e-books and ancillary materials for all these products. For each of our books,
the site contains companion Web sites that include frequently asked questions (FAQs),
code downloads, errata, updates, additional text and examples, additional self-test ques-
tions and new developments in programming languages and object-oriented programming
technologies. If you would like to learn more about the authors or Deitel & Associates, Inc.
please visit www.deitel.com. Good luck!

1.2 What Is a Computer?

A computer is a device capable of performing computations and making logical decisions at
speeds millions, even billions, of times faster than human beings can. For example, many of
today’s personal computers can perform hundreds of millions, even billions, of additions per
second. A person operating a desk calculator might require decades to complete the same
number of calculations a powerful personal computer can perform in one second. (Points to
ponder: How would you know whether the person added the numbers correctly? How would
you know whether the computer added the numbers correctly?) Today’s fastest supercom-
puters can perform hundreds of billions of additions per second—about as many calculations as
hundreds of thousands of people could perform in one year! And trillion-instruction-per-sec-
dond computers are already functioning in research laboratories!

Computers process data under the control of sets of instructions called computer pro-
grams. These programs guide the computer through orderly sets of actions specified by
people called computer programmers.

The various devices that comprise a computer system (such as the keyboard, screen, dis-
ks, memory and processing units) are referred to as hardware. The computer programs
that run on a computer are referred to as software. Hardware costs have been declining dra-
matically in recent years, to the point that personal computers have become a commodity.
Unfortunately, software-development costs have been rising steadily, as programmers
develop ever more powerful and complex applications without being able to improve sig-
nificantly the technology of software development. In this book, you will learn proven soft-
ware-development methods that can reduce software-development costs—top-down stepwise refinement, functionalization and object-oriented programming. Object-oriented
programming is widely believed to be the significant breakthrough that can greatly enhance
programmer productivity.

1.3 Computer Organization

Regardless of differences in physical appearance, virtually every computer may be envi-
ioned as being divided into six logical units or sections. These are as follows:
1. **Input unit.** This is the “receiving” section of the computer. It obtains information (data and computer programs) from input devices and places this information at the disposal of the other units so that the information may be processed. Most information is entered into computers today through typewriter-like keyboards, “mouse” devices and disks. In the future, most information will be entered by speaking to computers, by electronically scanning images and by video recording.

2. **Output unit.** This is the “shipping” section of the computer. It takes information processed by the computer and places it on various output devices to make the information available for use outside the computer. Information output from computers is displayed on screens, printed on paper, played through audio speakers, magnetically recorded on disks and tapes or used to control other devices.

3. **Memory unit.** This is the rapid-access, relatively low-capacity “warehouse” section of the computer. It retains information that has been entered through the input unit so that the information may be made immediately available for processing when it is needed. The memory unit also retains information that has already been processed until that information can be placed on output devices by the output unit. The memory unit often is called either memory, primary memory or random-access memory (RAM).

4. **Arithmetic and logic unit (ALU).** This is the “manufacturing” section of the computer. It is responsible for performing calculations such as addition, subtraction, multiplication and division. It contains the decision mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they are equal.

5. **Central processing unit (CPU).** This is the “administrative” section of the computer. It is the computer’s coordinator and is responsible for supervising the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be utilized in calculations and tells the output unit when to send information from the memory unit to certain output devices.

6. **Secondary storage unit.** This is the long-term, high-capacity “warehousing” section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (such as disks) until they are needed, possibly hours, days, months or even years later. Information in secondary storage takes longer to access than information in primary memory. The cost per unit of secondary storage is much less than the cost per unit of primary memory.

### 1.4 Evolution of Operating Systems

Early computers were capable of performing only one job or task at a time. This form of computer operation is often called single-user batch processing. The computer runs a single program at a time while processing data in groups or batches. In these early systems, users generally submitted their jobs to the computer center on decks of punched cards. Users often had to wait hours or even days before printouts were returned to their desks.

Software systems called operating systems were developed to help make it more convenient to use computers. Early operating systems managed the smooth transition between
jobs. This minimized the time it took for computer operators to switch between jobs and hence increased the amount of work, or throughput, computers could process.

As computers became more powerful, it became evident that single-user batch processing rarely utilized the computer’s resources efficiently. Instead, it was thought that many jobs or tasks could be made to share the resources of the computer to achieve better utilization. This is called multiprogramming. Multiprogramming involves the “simultaneous” operation of many jobs on the computer—the computer shares its resources among the jobs competing for its attention. With early multiprogramming operating systems, users still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several groups in industry and the universities pioneered timesharing operating systems. Timesharing is a special case of multiprogramming in which users access the computer through terminals, typically devices with keyboards and screens. In a typical timesharing computer system, there may be dozens or even hundreds of users sharing the computer at once. The computer does not actually run all the users’ jobs simultaneously. Rather, it runs a small portion of one user’s job and moves on to service the next user. The computer does this so quickly that it might provide service to each user several times per second. Thus the users’ programs appear to be running simultaneously. An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results, as with previous modes of computing. Also, if a particular user is currently idle, the computer can continue to service other users rather than wait for one user.

1.5 Personal, Distributed and Client/Server Computing

In 1977, Apple Computer popularized the phenomenon of personal computing. Initially, it was a hobbyist’s dream. Computers became economical enough for people to buy them for their own personal use. In 1981, IBM, the world’s largest computer vendor, introduced the IBM Personal Computer. Almost overnight, personal computing became legitimate in business, industry and government organizations.

But these computers were “stand-alone” units—people did their work on their own machines and transported disks back and forth to share information. Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in local area networks (LANs) within an organization. This led to the phenomenon of distributed computing, in which an organization’s computing, instead of being performed strictly at some central computer installation, is distributed over networks to the sites at which the real work of the organization is performed. Personal computers were powerful enough both to handle the computing requirements of individual users and to handle the basic communications tasks of passing information back and forth electronically.

Today’s most powerful personal computers are as powerful as the million-dollar machines of just a decade ago. The most powerful desktop machines—called workstations—provide individual users with enormous capabilities. Information is shared easily across computer networks where some computers called file servers offer a common store of programs and data that may be used by client computers distributed throughout the network (hence the term client/server computing). C and C++ have become and remain the languages of choice for writing operating systems. They also remain popular for writing computer networking, distributed client/server and Internet and Web applications, although
Java is now the dominant language in each of these areas. Many programmers have discovered that programming in Java helps them be more productive than programming in C or C++. Today’s popular operating systems, such as UNIX, Linux, MacOS, Windows and Windows 2000, provide the kinds of capabilities discussed in this section.

1.6 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others that require intermediate translation steps. Hundreds of computer languages are in use today. These may be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can directly understand only its own machine language. Machine language is the “natural language” of a particular computer. It is defined by the hardware design of that computer. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent (i.e., a particular machine language can be used on only one type of computer). Machine languages are cumbersome for humans, as can be seen by the following section of a machine-language program that adds overtime pay to base pay and stores the result in gross pay.

```
+1300042774
+1400593419
+1200274027
```

As computers became more popular, it became apparent that machine-language programming was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the elementary operations of computers. These English-like abbreviations formed the basis of assembly languages. Translator programs called assemblers were developed to convert assembly-language programs to machine language at computer speeds. The following section of an assembly-language program also adds overtime pay to base pay and stores the result in gross pay, but somewhat more clearly than its machine-language equivalent.

```
LOAD BASEPAY
ADD OVERPAY
STORE GROSSPAY
```

Although such code is clearer to humans, it is incomprehensible to computers until translated to machine language.

Computer usage increased rapidly with the advent of assembly languages, but programming in these still required many instructions to accomplish even the simplest tasks. To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks. The translator programs that convert high-level language programs into machine language are called compilers. High-
level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a statement such as

\[
grossPay = basePay + overTimePay
\]

Obviously, high-level languages are much more desirable from the programmer’s standpoint than either machine languages or assembly languages. C, C++ and Java are among the most powerful and most widely used high-level programming languages.

The process of compiling a high-level language program into machine language can take a considerable amount of computer time. Interpreter programs were developed to execute high-level language programs directly without the need for compiling those programs into machine language. Although compiled programs execute much faster than interpreted programs, interpreters are popular in program-development environments in which programs are recompiled frequently as new features are added and errors are corrected. Once a program is developed, a compiled version can be produced to run most efficiently. As we study Java, you will see that interpreters have played an especially important part in helping Java achieve its goal of portability across a great variety of platforms.

1.7 History of C++

C++ evolved from C, which evolved from two previous languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating-systems software and compilers. Ken Thompson modeled many features in his language B after their counterparts in BCPL and used B to create early versions of the UNIX operating system at Bell Laboratories in 1970 on a Digital Equipment Corporation PDP-7 computer. Both BCPL and B were “typeless” languages—every data item occupied one “word” in memory. For example, it was the programmer’s responsibility to treat a data item as a whole number or a real number.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972. C uses many important concepts of BCPL and B while adding data typing and other features. C initially became widely known as the development language of the UNIX operating system. Today, virtually all new major operating systems are written in C or C++. Over the past two decades, C has become available for most computers. C is hardware independent. With careful design, it is possible to write C programs that are portable to most computers.

By the late 1970s, C had evolved into what is now referred to as “traditional C,” or “Kernighan and Ritchie C.” The publication by Prentice Hall in 1978 of Kernighan and Ritchie’s book, The C Programming Language, brought wide attention to the language. This publication became one of the most successful computer science books ever.

The widespread use of C with various types of computers (sometimes called hardware platforms) led to many variations. These were similar, but often incompatible. This was a serious problem for programmers who needed to write portable programs that would run on several platforms. It became clear that a standard version of C was needed. In 1983, the X3J11 technical committee was created under the American National Standards Committee on Computers and Information Processing (X3) to “provide an unambiguous and machine-independent definition of the language.” In 1989, the standard was approved. ANSI cooperated with the International Standards Organization (ISO) to standardize C
worldwide; the joint standard document was published in 1990 and is referred to as ANSI/ISO 9899: 1990. The second edition of Kernighan and Ritchie, published in 1988, reflects this version called ANSI C, a version of the language still used worldwide (Ke88).

C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that “spruce up” the C language, but more importantly, it provides capabilities for object-oriented programming. C++ was also standardized by the ANSI and ISO committees.

There is a revolution brewing in the software community. Building software quickly, correctly and economically remains an elusive goal, and this at a time when demands for new and more powerful software are soaring. Objects are essentially reusable software components that model items in the real world. Software developers are discovering that using a modular, object-oriented design and implementation approach can make software-development groups much more productive than is possible with previous popular programming techniques such as structured programming. Object-oriented programs are often easier to understand, correct and modify.

Many other object-oriented languages have been developed, including Smalltalk, developed at Xerox’s Palo Alto Research Center (PARC). Smalltalk is a pure object-oriented language—literally everything is an object. C++ is a hybrid language—it is possible to program in either a C-like style, an object-oriented style or both.

1.8 History of Java

Perhaps the microprocessor revolution’s most important contribution to date is that it made possible the development of personal computers, which now number in the hundreds of millions worldwide. Personal computers have had a profound impact on people and the way organizations conduct and manage their business.

Many people believe that the next major area in which microprocessors will have a profound impact is in intelligent consumer-electronic devices. Recognizing this, Sun Microsystems funded an internal corporate research project code-named Green in 1991. The project resulted in the development of a C- and C++-based language that its creator, James Gosling, called Oak after an oak tree outside his window at Sun. It was later discovered that there already was a computer language called Oak. When a group of Sun people visited a local coffee place, the name Java was suggested, and it stuck.

The Green project ran into some difficulties. The marketplace for intelligent consumer-electronic devices was not developing as quickly as Sun had anticipated. Worse yet, a major contract for which Sun competed was awarded to another company. So the project was in danger of being canceled. By sheer good fortune, the World Wide Web exploded in popularity in 1993, and Sun people saw the immediate potential of using Java to create Web pages with so-called dynamic content. This breathed new life into the project.

Sun formally announced Java at a major conference in May 1995. Ordinarily, an event like this would not have generated much attention. However, Java generated immediate interest in the business community because of the phenomenal interest in the World Wide Web. Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of World Wide

Web servers (the computers that provide the content we see in our Web browsers), to provide applications for consumer devices (such as cell phones, pagers and personal digital assistants) and for many other purposes.

1.9 Java Class Libraries

Java programs consist of pieces called classes. Classes consist of pieces called methods that perform tasks and return information when they complete their tasks. You can program each piece you may need to form a Java program. However, most Java programmers take advantage of rich collections of existing classes in Java class libraries. The class libraries are also known as the Java APIs (Application Programming Interfaces). Thus, there are really two pieces to learning the Java “world.” The first is learning the Java language itself so that you can program your own classes; the second is learning how to use the classes in the extensive Java class libraries. Throughout the book, we discuss many library classes. Class libraries are provided primarily by compiler vendors, but many class libraries are supplied by independent software vendors (ISVs). Also, many class libraries are available from the Internet and World Wide Web as freeware or shareware. You can download freeware products and use them for free—subject to any restrictions specified by the copyright owner. You also can download shareware products for free, so you can try the software. Shareware products often are free of charge for personal use. However, for shareware products that you use regularly or use for commercial purposes, you are expected to pay a fee designated by the copyright owner.

Many freeware and shareware products are also open source. The source code for open-source products is freely available on the Internet, which enables you to learn from the source code, validate that the code serves its stated purpose and even modify the code. Often, open-source products require that you publish any enhancements you make so the open-source community can continue to evolve those products. One example of a popular open-source product is the Linux operating system.

Software Engineering Observation 1.1

Use a building-block approach to creating programs. Avoid reinventing the wheel. Use existing pieces—this is called software reuse and it is central to object-oriented programming.

[Note: We will include many of these Software Engineering Observations throughout the text to explain concepts that affect and improve the overall architecture and quality of software systems, and particularly, of large software systems. We will also highlight Good Programming Practices (practices that can help you write programs that are clearer, more understandable, more maintainable and easier to test and debug), Common Programming Errors (problems to watch out for so you do not make these same errors in your programs), Performance Tips (techniques that will help you write programs that run faster and use less memory), Portability Tips (techniques that will help you write programs that can run, with little or no modifications, on a variety of computers; these tips also include general observations about how Java achieves its high degree of portability), Testing and Debugging Tips (techniques that will help you remove bugs from your programs and, more important, techniques that will help you write bug-free programs to begin with) and Look and Feel Observations (techniques that will help you design the “look” and “feel” of your graphical user interfaces for appearance and ease of use). Many of these techniques and practices are only guidelines; you will, no doubt, develop your own preferred programming style.]
When programming in Java, you will typically use the following building blocks: Classes from class libraries, classes and methods you create yourself and classes and methods other people create and make available to you.

The advantage of creating your own classes and methods is that you know exactly how they work and you can examine the Java code. The disadvantage is the time-consuming and complex effort that goes into designing and developing new classes and methods.

**Performance Tip 1.1**
Using Java API classes and methods instead of writing your own versions can improve program performance, because these classes and methods are carefully written to perform efficiently. This technique also improves the prototyping speed of program development (i.e., the time it takes to develop a new program and get its first version running).

**Portability Tip 1.1**
Using classes and methods from the Java API instead of writing your own versions improves program portability, because these classes and methods are included in every Java implementation (assuming the same version number).

**Software Engineering Observation 1.3**
Extensive class libraries of reusable software components are available over the Internet and the Web. Many of these libraries provide source code and are available at no charge.

### 1.10 Other High-Level Languages

Hundreds of high-level languages have been developed, but only a few have achieved broad acceptance. **Fortran** (FORmula TRANslator) was developed by IBM Corporation between 1954 and 1957 to be used for scientific and engineering applications that require complex mathematical computations. Fortran is still widely used.

**COBOL** (COmmon Business Oriented Language) was developed in 1959 by a group of computer manufacturers and government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data. Today, about half of all business software is still programmed in COBOL. Approximately one million people are actively writing COBOL programs.

**Pascal** was designed at about the same time as C. It was created by Professor Nicklaus Wirth and was intended for academic use. We discuss Pascal further in the next section.

Basic was developed in 1965 at Dartmouth College as a simple language to help novices become comfortable with programming. **Bill Gates** implemented Basic on several early personal computers. Today, **Microsoft**—the company Bill Gates created—is the world’s leading software-development organization.

### 1.11 Structured Programming

During the 1960s, many large software-development efforts encountered severe difficulties. Software schedules were typically late, costs greatly exceeded budgets and the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity in the 1960s resulted
in the evolution of structured programming—a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify. Chapters 4 and 5 discuss the principles of structured programming.

One of the more tangible results of this research was the development of the Pascal programming language by Nicklaus Wirth in 1971. Pascal, named after the seventeenth-century mathematician and philosopher Blaise Pascal, was designed for teaching structured programming in academic environments and rapidly became the preferred programming language in most universities. Unfortunately, the language lacks many features needed to make it useful in commercial, industrial and government applications, so it has not been widely accepted in these environments.

The Ada programming language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of separate languages were being used to produce DOD’s massive command-and-control software systems. DOD wanted a single language that would fill most of its needs. Pascal was chosen as a base, but the final Ada language is quite different from Pascal. The language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is credited with writing the world’s first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada is called multitasking, which allows programmers to specify that many activities are to occur in parallel. The native capabilities of other widely used high-level languages we have discussed—including C and C++—generally allow the programmer to write programs that perform only one activity at a time. Java, through a technique we will explain called multithreading, also enables programmers to write programs with parallel activities. [Note: Most operating systems provide libraries specific to individual platforms (sometimes called platform-dependent libraries) that enable high-level languages like C and C++ to specify that many activities are to occur in parallel in a program.]

1.12 The Internet and the World Wide Web

The Internet was developed more than three decades ago with funding supplied by the Department of Defense. Originally designed to connect the main computer systems of about a dozen universities and research organizations, the Internet today is accessible by hundreds of millions of computers worldwide.

With the introduction of the World Wide Web—which allows computer users to locate and view multimedia-based documents on almost any subject—the Internet has exploded into one of the world’s premier communication mechanisms.

The Internet and the World Wide Web will surely be listed among the most important and profound creations of humankind. In the past, most computer applications ran on computers that were not connected to one another. Today’s applications can be written to communicate among the world’s hundreds of millions of computers. The Internet mixes computing and communications technologies. It makes our work easier. It makes information instantly and conveniently accessible worldwide. It makes it possible for individuals and local small businesses to get worldwide exposure. It is changing the nature of the way business is done. People can search for the best prices on virtually any product or service. Special-interest communities can stay in touch with one another. Researchers can be made instantly aware of the latest breakthroughs worldwide.
Java How to Program: Fourth Edition presents programming techniques that allow Java applications to use the Internet and World Wide Web to interact with other applications. These capabilities, and the capabilities discussed in our companion book Advanced Java 2 Platform How to Program, allow Java programmers to develop the kind of enterprise-level distributed applications that are used in industry today. Java applications can be written to execute on any computer platform, yielding major savings in systems development time and cost for corporations. If you have been hearing a great deal about the Internet and World Wide Web lately, and if you are interested in developing applications to run over the Internet and the Web, learning Java may be the key to challenging and rewarding career opportunities for you.

1.13 Basics of a Typical Java Environment

Java systems generally consist of several parts: An environment, the language, the Java Applications Programming Interface (API) and various class libraries. The following discussion explains a typical Java program development environment, as shown in Fig. 1.1.

Java programs normally go through five phases to be executed (Fig. 1.1). These are: edit, compile, load, verify and execute. We discuss these concepts in the context of the Java 2 Software Development Kit (J2SDK) that is included on the CD that accompanies this book. Carefully follow the installation instructions for the J2SDK provided on the CD to ensure that you set up your computer properly to compile and execute Java programs. [Note: If you are not using UNIX/Linux, Windows 95/98/ME or Windows NT/2000, refer to the manuals for your system’s Java environment or ask your instructor how to accomplish these tasks in your environment (which will probably be similar to the environment in Fig. 1.1).]

Phase 1 consists of editing a file. This is accomplished with an editor program (normally known as an editor). The programmer types a Java program, using the editor, and makes corrections, if necessary. When the programmer specifies that the file in the editor should be saved, the program is stored on a secondary storage device, such as a disk. Java program file names end with the .java extension. Two editors widely used on UNIX/Linux systems are vi and emacs. On Windows 95/98/ME and Windows NT/2000, simple edit programs like the DOS Edit command and the Windows Notepad will suffice. Java integrated development environments (IDEs), such as Forté for Java Community Edition, NetBeans, Borland’s JBuilder, Symantec’s Visual Cafe and IBM’s VisualAge have built-in editors that are integrated into the programming environment. We assume the reader knows how to edit a file.

[Note that Forté for Java Community Edition is written in Java and is free for non-commercial use. It is included on the CD accompanying this book. Sun updates this software approximately twice a year. Newer versions can be downloaded from

www.sun.com/forte/ffj

Forté for Java Community Edition executes on most major platforms. This book is written for any generic Java 2 development environment. It is not dependent on Forté for Java Community Edition. Our example programs should operate properly with most Java integrated development environments.]

In Phase 2 (discussed again in Chapters 2 and 3), the programmer gives the command javac to compile the program. The Java compiler translates the Java program into byte-
codes—the language understood by the Java interpreter. To compile a program called Welcome.java, type

![Typical Java environment](image)

**Fig. 1.1** Typical Java environment.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/8/01
javac Welcome.java

at the command window of your system (i.e., the MS-DOS prompt in Windows, the Command Prompt in Windows NT/2000 or the shell prompt in UNIX/Linux). If the program compiles correctly, the compiler produces a file called Welcome.class. This is the file containing the bytecodes that will be interpreted during the execution phase.

Phase 3 is called loading. The program must first be placed in memory before it can be executed. This is done by the class loader, which takes the .class file (or files) containing the bytecodes and transfers it to memory. The .class file can be loaded from a disk on your system or over a network (such as your local university or company network or even the Internet). There are two types of programs for which the class loader loads .class files—

applications and applets. An application is a program (such as a word-processor program, a spreadsheet program, a drawing program or an e-mail program) that normally is stored and executed from the user’s local computer. An applet is a small program that normally is stored on a remote computer that users connect to via a World Wide Web browser. Applets are loaded from a remote computer into the browser, executed in the browser and discarded when execution completes. To execute an applet again, the user must point a browser at the appropriate location on the World Wide Web and reload the program into the browser.

Applications are loaded into memory and executed by using the Java interpreter via the command java. When executing a Java application called Welcome, the command

java Welcome

invokes the interpreter for the Welcome application and causes the class loader to load information used in the Welcome program. [Note: Many Java programmers refer to the interpreter as the Java Virtual Machine or the JVM.]

The class loader also executes when a World Wide Web browser such as Netscape Navigator or Microsoft Internet Explorer loads a Java applet. Browsers are used to view documents on the World Wide Web called Hypertext Markup Language (HTML) documents. HTML describes the format of a document in a manner that is understood by the browser application (we introduce HTML in Section 3.4; for a detailed treatment of HTML and other Internet programming technologies, please see our text Internet and World Wide Web How to Program, Second Edition). An HTML document may refer to a Java applet. When the browser sees an applet referenced in an HTML document, the browser launches the Java class loader to load the applet (normally from the location where the HTML document is stored). Each browser that supports Java has a built-in Java interpreter. After the applet loads, the browser’s Java interpreter executes the applet. Applets can also execute from the command line, using the appletviewer command provided with the J2SDK—the set of tools including the compiler (javac), interpreter (java), appletviewer and other tools used by Java programmers. Like Netscape Navigator and Microsoft Internet Explorer, the appletviewer requires an HTML document to invoke an applet. For example, if the Welcome.html file refers to the Welcome applet, the appletviewer command is used as follows:

appletviewer Welcome.html

This causes the class loader to load the information used in the Welcome applet. The appletviewer is a minimal browser—it knows only how to interpret references to applets and ignores all other HTML in a document.
Before the Java interpreter built into a browser or the appletviewer executes the bytecodes in an applet, the bytecodes are verified by the bytecode verifier in Phase 4. This ensures that the bytecodes for classes that are loaded from the Internet (referred to as downloaded classes) are valid and that they do not violate Java’s security restrictions. Java enforces strong security, because Java programs arriving over the network should not be able to cause damage to your files and your system (as computer viruses might). Note that bytecode verification also occurs in applications that download classes from a network.

Finally, in Phase 5, the computer, under the control of its CPU, interprets the program one bytecode at a time, thus performing the actions specified by the program.

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we will discuss in this text. For example, an executing program might attempt to divide by zero (an illegal operation in Java just as it is in arithmetic). This would cause the Java program to print an error message. The programmer would return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections work properly.

**Common Programming Error 1.1**

Errors like division-by-zero errors occur as a program runs, so these errors are called runtime errors or execution-time errors. Fatal runtime errors cause programs to terminate immediately without having successfully performed their jobs. Nonfatal runtime errors allow programs to run to completion, often producing incorrect results.

Most programs in Java input or output data. When we say that a program prints a result, we normally mean that the program displays results on the computer screen. Data may be output to other devices, such as disks and hardcopy printers.

### 1.14 General Notes about Java and This Book

Java is a powerful language. Experienced programmers sometimes take pride in being able to create some weird, contorted, convoluted usage of a language. This is a poor programming practice. It makes programs more difficult to read, more likely to behave strangely, more difficult to test and debug and more difficult to adapt to changing requirements. This book is also geared for novice programmers, so we stress clarity. The following is our first “good programming practice.”

**Good Programming Practice 1.1**

Write your Java programs in a simple and straightforward manner. This is sometimes referred to as KIS (“keep it simple”). Do not “stretch” the language by trying bizarre usages.

You have heard that Java is a portable language and that programs written in Java can run on many different computers. For programming in general, portability is an elusive goal. For example, the ANSI C standard document\(^2\) contains a lengthy list of portability issues, and complete books have been written that discuss portability.\(^3,4\)

---

Portability Tip 1.2

Although it is easier to write portable programs in Java than in other programming languages, there are differences among compilers, interpreters and computers that can make portability difficult to achieve. Simply writing programs in Java does not guarantee portability. The programmer will occasionally need to deal with compiler and computer variations.

Testing and Debugging Tip 1.1

Always test your Java programs on all systems on which you intend to run those programs, to ensure that your Java programs will work correctly for their intended audience.

We have done a careful walkthrough of Sun’s Java documentation and audited our presentation against it for completeness and accuracy. However, Java is a rich language, and there are some subtleties in the language and some topics we have not covered. If you need additional technical details on Java, we suggest that you read the most current Java documentation available over the Internet at java.sun.com. Our book contains an extensive bibliography of books and papers on the Java language in particular and on object-oriented programming in general. A Web-based version of the Java API documentation can be found at java.sun.com/j2se/1.3/docs/api/index.html. Also, you can download this documentation to your own computer from java.sun.com/j2se/1.3/docs.html.

Good Programming Practice 1.2

Read the documentation for the version of Java you are using. Refer to this documentation frequently to be sure you are aware of the rich collection of Java features and that you are using these features correctly.

Good Programming Practice 1.3

Your computer and compiler are good teachers. If, after carefully reading your Java documentation manual, you are not sure how a feature of Java works, experiment and see what happens. Study each error or warning message you get when you compile your programs, and correct the programs to eliminate these messages.

Good Programming Practice 1.4

The Java 2 Software Development Kit comes with the Java source code. Many programmers read the actual source code of the Java API classes to determine how those classes work and to learn additional programming techniques. If the Java API documentation is not clear on a particular topic, try studying the source code of the class.

In this book, we explain how Java works in its current implementations. Perhaps the most striking problem with the early versions of Java is that Java programs execute interpretively on the client’s machine. Interpreters execute slowly compared to fully compiled machine code.

Performance Tip 1.2

Interpreters have an advantage over compilers for the Java world, namely that an interpreted program can begin execution immediately as soon as it is downloaded to the client’s machine, whereas a source program to be compiled must first suffer a potentially long delay as the program is compiled before it can be executed.
Portability Tip 1.3

Although only Java interpreters were available to execute bytecodes at the client’s site on early Java systems, compilers that translate Java bytecodes (or in some cases the Java source code) into the native machine code of the client’s machine have been written for most popular platforms. These compiled programs perform comparably to compiled C or C++ code. However, there are not bytecode compilers for every Java platform, so Java programs will not perform at the same level on all platforms.

Applets present some more interesting issues. Remember, an applet could be coming from virtually any Web server in the world. So the applet will have to be able to run on any possible Java platform.

Portability Tip 1.4

Short, fast-executing Java applets can certainly still be interpreted. But what about more substantial, compute-intensive applets? Here, the user might be willing to suffer the compilation delay to get better execution performance. For some especially performance-intensive applets, the user might have no choice; interpreted code would run too slowly for the applet to perform properly, so the applet would have to be compiled.

Portability Tip 1.5

An intermediate step between interpreters and compilers is a just-in-time (JIT) compiler that, as the interpreter runs, produces compiled code for the programs and executes the programs in machine language rather than reinterpreting them. JIT compilers do not produce machine language that is as efficient as that from a full compiler.

Portability Tip 1.6

For the latest information on high-speed Java program translation, you might want to read about Sun’s HotSpot™ compiler, so visit java.sun.com/products/hotspot. The HotSpot compiler is a standard component of the Java 2 Software Development Kit.

The Java compiler, `javac`, is not a traditional compiler in that it does not convert a Java program from source code into native machine code for a particular computer platform. Instead, the Java compiler translates source code into bytecodes. Bytecodes are the language of the Java Virtual Machine—a program that simulates the operation of a computer and executes its own machine language (i.e., Java bytecodes). The Java Virtual Machine is implemented in the J2SDK as the `java` interpreter, which translates the bytecodes into native machine language for the local computer platform.

Software Engineering Observation 1.4

For organizations wanting to do heavy-duty information systems development, Integrated Development Environments (IDEs) are available from many major software suppliers, including Sun Microsystems. The IDEs provide many tools for supporting the software-development process, such as editors for writing and editing programs, debuggers for locating logic errors in programs and many other features.

Software Engineering Observation 1.5

Sun Microsystems, Inc.’s powerful Java IDE—Forté for Java, Community Edition—is available on the CD that accompanies this book and can be downloaded from www.sun.com/forte/ffj.
1.15 Thinking About Objects: Introduction to Object Technology and the Unified Modeling Language

Now we begin our early introduction to object orientation. We will see that object orientation is a natural way of thinking about the world and a natural way of writing computer programs.

In the bodies of each of the first seven chapters, we concentrate on the “conventional” methodology of structured programming, because the objects we will build will be composed in part of structured-program pieces. However, we end each chapter with a “Thinking About Objects” section in which we present a carefully paced introduction to object orientation. Our goal in these “Thinking About Objects” sections is to help you develop an object-oriented way of thinking, so that you immediately can use the object-oriented programming techniques that we present starting in Chapter 8. The “Thinking About Objects” sections also introduce you to the *Unified Modeling Language (UML)*. The UML is a graphical language that allows people who build systems (e.g., software architects, systems engineers, programmers and so on) to represent their object-oriented designs, using a common notation.

In this section, we introduce basic concepts (i.e., “object think”) and terminology (i.e., “object speak”). Chapters 2–13, 15 and 22 and Appendices G–I include optional “Thinking About Objects” sections that present a substantial case study that applies the techniques of *object-oriented design (OOD)*. The optional sections at the ends of Chapters 2 through 7 analyze a typical problem statement that requires a system to be built, determine the objects required to implement that system, determine the attributes the objects will have, determine the behaviors these objects will exhibit and specify how the objects will interact with one another to meet the system requirements. All this occurs before you learn to write object-oriented Java programs! The optional sections at the ends of Chapters 8–13 and 15 modify and enhance the design presented in Chapters 2–7. Chapter 22 presents how to display our multimedia-rich design on the screen. The optional “Thinking About Objects” sections in each chapter apply the concepts discussed in that chapter to the case study. In Appendices G, H and I, we present a complete Java implementation of the object-oriented system we design in the earlier chapters.

This case study will help prepare you for the kinds of substantial projects you are likely to encounter in industry. If you are a student and your instructor does not plan to include this case study in your course, you may want to cover it on your own time. We believe it will be well worth your time to walk through this large and challenging project, because the material presented in the case-study sections reinforces the material covered in the corresponding chapters. You will experience a solid introduction to object-oriented design with the UML. Also, you will sharpen your code-reading skills by touring a carefully written and well-documented 3,594-line Java program that completely solves the problem presented in the case study.

We begin our introduction to object orientation with some key terminology. Everywhere you look in the real world you see them—*objects*: People, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. We possess the marvelous ability of *abstraction*, which enables us to view screen images such as people, planes, trees and mountains as objects, rather than as individual dots of color (called *pixels*—for “picture elements”). We can, if we wish, think in terms of beaches rather than grains of sand, forests rather than trees and houses rather than bricks.
We might be inclined to divide objects into two categories—animate objects and inanimate objects. Animate objects are “alive” in some sense; they move around and do things. Inanimate objects, on the other hand, seem not to do much at all. They do not move on their own. All these objects, however, do have some things in common. They all have attributes like size, shape, color, and weight, and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water).

Humans learn about objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Cars, trucks, little red wagons and roller skates have much in common.

Object-oriented design models real-world objects. It takes advantage of class relationships, where objects of a certain class—such as a class of vehicles—have the same characteristics. It takes advantage of inheritance relationships, and even multiple-inheritance relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class “convertible” certainly has the characteristics of the more general class “automobile,” plus a convertible’s roof goes up and down.

Object-oriented design provides a more natural and intuitive way to view the design process—namely, by modeling real-world objects, their attributes, their behavior. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commanding a soldier to stand at attention), objects also communicate via messages.

OOD encapsulates data (attributes) and functions (behavior) into objects; the data and functions of an object are intimately tied together. Objects have the property of information hiding. This means that, although objects may know how to communicate with one another across well-defined interfaces, objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. Surely, it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

Languages such as Java are object-oriented—programming in such a language is called object-oriented programming (OOP) and allows designers to implement the object-oriented design as a working system. Languages such as C, on the other hand, are procedural programming languages, so programming tends to be action-oriented. In C, the unit of programming is the function. In Java, the unit of programming is the class from which objects are eventually instantiated (a fancy term for “created”). Java classes contain methods (that implement class behaviors) and attributes (that implement class data).

C programmers concentrate on writing functions. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. Data are certainly important in C, but the view is that data exist primarily in support of the actions that functions perform. The verbs in a system specification help the C programmer determine the set of functions needed to implement that system.

Java programmers concentrate on creating their own user-defined types called classes and components. Each class contains data and the set of functions that manipulate that data.

---

5. We will learn later that although Java—unlike C++—does not support multiple inheritance, it does offer most of the key benefits of this technology by supporting multiple interfaces per class.
The data components of a Java class are called attributes. The function components of a Java class are called methods. Just as an instance of a built-in type such as int is called a variable, an instance of a user-defined type (i.e., a class) is called an object. The programmer uses built-in types as the “building blocks” for constructing user-defined types. The focus in Java is on classes (out of which we make objects) rather than on functions. The nouns in a system specification help the Java programmer determine the set of classes from which objects will be created that will work together to implement the system.

Classes are to objects as blueprints are to houses. We can build many houses from one blueprint, and we can instantiate many objects from one class. Classes can also have relationships with other classes. For example, in an object-oriented design of a bank, the “bank-teller” class needs to relate to the “customer” class. These relationships are called associations.

We will see that, when software is packaged as classes, these classes can be reused in future software systems. Groups of related classes are often packaged as reusable components. Just as real-estate brokers tell their clients that the three most important factors affecting the price of real estate are “location, location and location,” many people in the software community believe that the three most important factors affecting the future of software development are “reuse, reuse and reuse.”

Indeed, with object technology, we can build much of the software we will need by combining “standardized, interchangeable parts” called classes. This book teaches you how to “craft valuable classes” for reuse. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software-development efforts—an exciting possibility.

**Introduction to Object-Oriented Analysis and Design (OOAD)**

You soon will be writing programs in Java. How will you create the code for your programs? If you are like many beginning programmers, you will simply turn on your computer and start typing. This approach may work for small projects, but what would you do if you were asked to create a software system to control the automated teller machines for a major bank? Such a project is too large and complex for you to sit down and simply start typing.

To create the best solutions, you should follow a detailed process for obtaining an analysis of your project’s requirements and developing a design for satisfying those requirements. Ideally, you would go through this process and have its results reviewed and approved by your superiors before writing any code for your project. If this process involves analyzing and designing your system from an object-oriented point of view, we call it an object-oriented analysis and design (OOAD) process. Experienced programmers know that, no matter how simple a problem appears, time spent on analysis and design can save innumerable hours that might be lost from abandoning an ill-planned system-development approach part of the way through its implementation.

OOAD is the generic term for the ideas behind the process we employ to analyze a problem and develop an approach for solving it. Small problems like the ones discussed in these first few chapters do not require an exhaustive process. It may be sufficient to write pseudocode before we begin writing code. (Pseudocode is an informal means of expressing program code. It is not actually a programming language, but we can use it as a kind of “outline” to guide us as we write our code. We introduce pseudocode in Chapter 4.)

Pseudocode can suffice for small problems, but as problems and the groups of people solving these problems increase in size, the methods of OOAD become more involved. Ide-
ally, a group should agree on a strictly defined process for solving the problem and on a uniform way of communicating the results of that process to one another. Although many different OOAD processes exist, a single graphical language for communicating the results of any OOAD process has become widely used. This language is known as the Unified Modeling Language (UML). The UML was developed in the mid-1990s under the initial direction of three software methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson.

**History of the UML**

In the 1980s, increasing numbers of organizations began using OOP to program their applications, and a need developed for an established process with which to approach OOAD. Many methodologists—including Booch, Rumbaugh and Jacobson—individually produced and promoted separate processes to satisfy this need. Each of these processes had its own notation, or “language” (in the form of graphical diagrams), to convey the results of analysis and design.

By the early 1990s, different companies, and even different divisions within the same company, were using different processes and notations. Additionally, these companies wanted to use software tools that would support their particular processes. With so many processes, software vendors found it difficult to provide such tools. Clearly, a standard notation and standard processes were needed.

In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation, and the two began working to unify their popular processes. They were soon joined by Ivar Jacobson. In 1996, the group released early versions of the UML to the software engineering community and requested feedback. Around the same time, an organization known as the Object Management Group™ (OMG™) invited submissions for a common modeling language. The OMG is a not-for-profit organization that promotes the use of object-oriented technology by issuing guidelines and specifications for object-oriented technologies. Several corporations—among them HP, IBM, Microsoft, Oracle and Rational Software—had already recognized the need for a common modeling language. These companies formed the UML Partners in response to the OMG’s request for proposals. This consortium developed the UML version 1.1 and submitted it to the OMG. The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML. In 2001, the OMG released the UML version 1.4 (the current version at the time this book was published) and is working on version 2.0 (scheduled tentatively for release in 2002).

**What is the UML?**

The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems. It has indeed unified the various popular notational schemes. Those who design systems use the language (in the form of graphical diagrams) to model their systems.

An attractive feature of the UML is its flexibility. The UML is extendable and is independent of the many OOAD processes. UML modelers are free to develop systems by using various processes, but all developers can now express those systems with one standard set of notations.

The UML is a complex, feature-rich graphical language. In our “Thinking About Objects” sections, we present a concise, simplified subset of these features. We then use this subset to guide the reader through a first design experience with the UML intended for the

Object-oriented technology is ubiquitous in the software industry, and the UML is rapidly becoming so. Our goal in these “Thinking About Objects” sections is to encourage you to think in an object-oriented manner as early, and as often, as possible. In the “Thinking About Objects” section at the end of Chapter 2, you will begin to apply object technology to implement a solution to a substantial problem. We hope that you will find this optional project to be an enjoyable and challenging introduction to object-oriented design with the UML and to object-oriented programming.

### 1.16 Discovering Design Patterns: Introduction

This section begins our treatment of design patterns, entitled “Discovering Design Patterns.” Most of the examples provided in this book 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 introductory programming concepts. However, some programs, such as our optional elevator-simulation case study, are more complex—they can require thousands of lines of code or even more, contain many interactions among objects and involve many user interactions. Larger systems, such as automated teller machines or air-traffic control systems, could contain 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. Using design patterns can substantially reduce the complexity of the design process. Designing an ATM system will be a somewhat less formidable task if developers use design patterns. In addition, well-designed object-oriented software allows designers to reuse and integrate preexisting components in future systems. Design patterns benefit system developers by

- helping to construct reliable software using proven architectures and accumulated industry expertise
- promoting design reuse in future systems
- helping to identify common mistakes and pitfalls that occur when building systems
- helping to design systems independently of the language in which they will ultimately be implemented

---

6. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. (Massachusetts: Addison-Wesley, 1995).
Chapter 1 Introduction to Computers, the Internet and the Web 27

• 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 a set of 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.

In software, design patterns are neither classes nor objects. Rather, designers use design patterns to construct 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. In this chapter, we discuss fundamental object-oriented design patterns and architectures, as well as their importance in constructing well-engineered software.

We present several design patterns in Java, but these design patterns can be implemented in any object-oriented language, such as C++ or Visual Basic. We describe several design patterns used by Sun Microsystems in the Java API. We use design patterns in many programs in this book, which we will identify throughout our discussion. These programs provide examples of using design patterns to construct reliable, robust object-oriented software.

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”—used their combined expertise to write the book Design Patterns, Elements of Reusable Object-Oriented Software (Addison-Wesley: 1995). This book described 23 design patterns, each providing a solution to a common software design problem in industry. The book groups design patterns into three categories—creation design patterns, structural design patterns and behavioral design patterns. Creation design patterns describe techniques to instantiate objects (or groups of objects). Structural design patterns allow designers to organize classes and objects into larger structures. Behavioral design patterns assign responsibilities to objects.

The gang-of-four book showed that design patterns evolved naturally through years of industry experience. In his article Seven Habits of Successful Pattern Writers,7 John Vlissides states that “the single most important activity in pattern writing is reflection.” This statement implies that, to create patterns, developers must reflect on, and document, their successes (and mistakes). Developers use design patterns to capture and employ this collective industry experience, which ultimately helps them avoid making the same mistakes twice.

New design patterns are being created all the time and being introduced rapidly to designers worldwide via the Internet. The topic of design patterns has generally been viewed as advanced, but authors such as ourselves are working this material into introductory and intermediate-level textbooks to help make this important knowledge available to a much wider audience.

Our treatment of design patterns begins with this required section in Chapter 1 and continues with five optional “Discovering Design Patterns” sections at the ends of Chapters 9, 13, 15, 17 and 21. Each of these sections is placed at the end of the chapter that introduces

the necessary Java technologies. If you are a student and your instructor does not plan to include this material in your course, we encourage you to read this material on your own.

1.17 Tour of the Book

You are about to study one of today’s most exciting and rapidly developing computer programming languages. Mastering Java will help you develop powerful business and personal computer-applications software. In this section, we take a tour of the many capabilities of Java you will study in *Java How to Program: Fourth Edition*.

**Chapter 1—Introduction to Computers, the Internet and the Web**—discusses what computers are, how they work and how they are programmed. The chapter gives a brief history of the development of programming languages from machine languages, to assembly languages, to high-level languages. The origin of the Java programming language is discussed. The chapter includes an introduction to a typical Java programming environment. The chapter also introduces object technology, the Unified Modeling Language and design patterns.

**Chapter 2—Introduction to Java Applications**—provides a lightweight introduction to programming applications in the Java programming language. The chapter introduces nonprogrammers to basic programming concepts and constructs. The programs in this chapter illustrate how to display (also called *outputting*) data on the screen to the user and how to obtain (also called *inputting*) data from the user at the keyboard. Some of the input and output is by performed using a *graphical user interface (GUI)* component called `JOptionPane` that provides predefined windows (called dialog boxes) for input and output. This allows a nonprogrammer to concentrate on fundamental programming concepts and constructs rather than on the more complex GUI event handling. Using `JOptionPane` here enables us to delay our introduction of GUI event handling to Chapter 6, “Methods.” Chapter 2 also provides detailed treatments of *decision making* and *arithmetic operations*. After studying this chapter, the student will understand how to write simple, but complete, Java applications.

**Chapter 3—Introduction to Java Applets**—introduces another type of Java program, called an *applet*. Applets are Java programs designed to be transported over the Internet and executed in World Wide Web browsers (like Netscape Navigator and Microsoft Internet Explorer). The chapter introduces applets, using several of the demonstration applets supplied with the Java 2 Software Development Kit (J2SDK). We use `appletviewer` (a utility supplied with the J2SDK) or a Web browser to execute several sample applets. We then write Java applets that perform tasks similar to the programs of Chapter 2, and we explain the similarities and differences between applets and applications. After studying this chapter, the student will understand how to write simple, but complete, Java applets. The next several chapters use both applets and applications to demonstrate additional key programming concepts.

**Chapter 4—Control Structures: Part 1**—focuses on the program-development process. The chapter discusses how to take a *problem statement* (i.e., a *requirements document*) and from it develop a working Java program, including performing intermediate steps in pseudocode. The chapter introduces some fundamental data types and simple control structures used for decision making (*if* and *if/else*) and repetition (*while*). We examine counter-controlled repetition and sentinel-controlled repetition, and introduce Java’s increment, decrement and assignment operators. The chapter uses simple flowcharts...
to show the flow of control through each of the control structures. The techniques discussed in Chapters 2 through 7 constitute a large part of what has been traditionally taught in the universities under the topic of structured programming. With Java, we do object-oriented programming. In doing so, we discover that the insides of the objects we build make abundant use of control structures. We have had a particularly positive experience assigning problems 4.11 through 4.14 in our introductory courses. Since these four problems have similar structure, doing all four is a nice way for students to “get the hang of” the program-development process. This chapter helps the student develop good programming habits in preparation for dealing with the more substantial programming tasks in the remainder of the text.

Chapter 5—Control Structures: Part 2—continues the discussions of Java control structures (for, the switch selection structure and the do/while repetition structure). The chapter explains the labeled break and continue statements with live-code examples. The chapter also contains a discussion of logical operators—&& (logical AND), & (boolean logical AND), || (logical OR), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR) and ! (NOT). There is a substantial exercise set including mathematical, graphical and business applications. Students will enjoy Exercise 5.25, which asks them to write a program with repetition and decision structures that prints the iterative song, “The Twelve Days of Christmas.” The more mathematically inclined students will enjoy problems on binary, octal, decimal and hexadecimal number systems, calculating the mathematical constant π with an infinite series, Pythagorean triples and De Morgan’s Laws. Our students particularly enjoy the challenges of triangle-printing and diamond-printing in Exercises 5.10, 5.18 and 5.20; these problems help students learn to deal with nested repetition structures—a complex topic to master in introductory courses.

Chapter 6—Methods—takes a deeper look inside objects. Objects contain data called instance variables and executable units called methods (these are often called functions in non-object-oriented procedural programming languages like C and member functions in C++). We explore methods in depth and include a discussion of methods that “call themselves,” so-called recursive methods. We discuss class-library methods, programmer-defined methods and recursion. The techniques presented in Chapter 6 are essential to the production of properly structured programs, especially the kinds of larger programs and software that system programmers and application programmers are likely to develop in real-world applications. The “divide and conquer” strategy is presented as an effective means for solving complex problems by dividing them into simpler interacting components. Students enjoy the treatment of random numbers and simulation, and they appreciate the discussion of the dice game of craps that makes elegant use of control structures (this is one of our most successful lectures in our introductory courses). The chapter offers a solid introduction to recursion and includes a table summarizing the dozens of recursion examples and exercises distributed throughout the remainder of the book. Some texts leave recursion for a chapter late in the book; we feel this topic is best covered gradually throughout the text. The topic of method overloading (i.e., allowing multiple methods to have the same name as long as they have different “signatures”) is motivated and explained clearly. We introduce events and event handling—elements required for programming graphical user interfaces. Events are notifications of state change such as button clicks, mouse clicks and pressing a keyboard key. Java allows programmers to specify the responses to events by coding methods called event handlers.
The extensive collection of exercises at the end of the chapter includes several classical recursion problems such as the Towers of Hanoi; we revisit this problem later in the text where we employ graphics, animation and sound to make the problem “come alive.” There are many mathematical and graphical examples. Our students particularly enjoy the development of a “Computer-Assisted Instruction” system in Exercises 6.31 through 6.33; we ask students to develop a multimedia version of this system later in the book. Students will enjoy the challenges of the “mystery programs.” The more mathematically inclined students will enjoy problems on perfect numbers, greatest common divisors, prime numbers and factorials.

**Chapter 7—Arrays**—explores the processing of data in lists and tables of values. Arrays in Java are processed as objects, further evidence of Java’s commitment to almost 100% object orientation. We discuss the structuring of data into arrays, or groups, of related data items of the same type. The chapter presents numerous examples of both single-subscripted arrays and double-subscripted arrays. It is widely recognized that structuring data properly is just as important as using control structures effectively in the development of properly structured programs. Examples in the chapter investigate various common array manipulations, printing histograms, sorting data, passing arrays to methods and an introduction to the field of survey data analysis (with simple statistics). A feature of this chapter is the discussion of elementary sorting and searching techniques and the presentation of binary searching as a dramatic improvement over linear searching. The end-of-chapter exercises include a variety of interesting and challenging problems, such as improved sorting techniques, the design of an airline reservations system, an introduction to the concept of turtle graphics (made famous in the LOGO programming language) and the Knight’s Tour and Eight Queens problems that introduce the notions of heuristic programming so widely employed in the field of artificial intelligence. The exercises conclude with a series of recursion problems including the selection sort, palindromes, linear search, binary search, the eight queens, printing an array, printing a string backwards and finding the minimum value in an array. The chapter exercises include a delightful simulation of the classic race between the tortoise and the hare, card shuffling and dealing algorithms, recursive quicksort and recursive maze traversals. A special section entitled “Building Your Own Computer” explains machine-language programming and proceeds with the design and implementation of a computer simulator that allows the reader to write and run machine language programs. This unique feature of the text will be especially useful to the reader who wants to understand how computers really work. Our students enjoy this project and often implement substantial enhancements; many enhancements are suggested in the exercises. In Chapter 19, another special section guides the reader through building a compiler; the machine language produced by the compiler is then executed on the machine language simulator produced in Chapter 7. Information is communicated from the compiler to the simulator in sequential files (presented in Chapter 16).

**Chapter 8—Object-Based Programming**—begins our deeper discussion of classes. The chapter represents a wonderful opportunity for teaching data abstraction the “right way”—through a language (Java) expressly devoted to implementing abstract data types (ADTs). The chapter focuses on the essence and terminology of classes and objects. What is an object? What is a class of objects? What does the inside of an object look like? How are objects created? How are they destroyed? How do objects communicate with one another? Why are classes such a natural mechanism for packaging software as reusable
componentry? The chapter discusses implementing ADTs as Java-style classes, accessing class members, enforcing information hiding with `private` instance variables, separating interface from implementation, using access methods and utility methods and initializing objects with constructors (and using overloaded constructors). The chapter discusses declaring and using constant references, `composition`—the process of building classes that have as members references to objects, the `this` reference that enables an object to “know itself,” dynamic memory allocation, `static` class members for containing and manipulating class-wide data and examples of popular abstract data types such as stacks and queues. The chapter introduces the `package` statement and discusses how to create reusable packages. The chapter also introduces creating `Java archive (JAR)` files and demonstrates how to use JAR files to deploy applets that consist of multiple classes. The chapter exercises challenge the student to develop classes for complex numbers, rational numbers, times, dates, rectangles, huge integers, a class for playing Tic-Tac-Toe, a savings-account class and a class for holding sets of integers.

Chapter 9—Object-Oriented Programming—discusses the relationships among classes of objects and programming with related classes. How can we exploit commonality between classes of objects to minimize the amount of work it takes to build large software systems? What is polymorphism? What does it mean to “program in the general” rather than “program in the specific?” How does programming in the general make it easy to modify systems and add new features with minimal effort? How can we program for a whole category of objects rather than programming individually for each type of object? The chapter deals with one of the most fundamental capabilities of object-oriented programming languages, inheritance, which is a form of software reusability in which new classes are developed quickly and easily by absorbing the capabilities of existing classes and adding appropriate new capabilities. The chapter discusses the notions of superclasses and subclasses, `protected` members, direct superclasses, indirect superclasses, use of constructors in superclasses and subclasses, and software engineering with inheritance. This chapter introduces inner classes that help hide implementation details. Inner classes are most frequently used to create GUI event handlers. Named inner classes can be declared inside other classes and are useful in defining common event handlers for several GUI components. Anonymous inner classes are declared inside methods and are used to create one object—typically an event handler for a specific GUI component. The chapter compares inheritance (“is a” relationships) with composition (“has a” relationships). A feature of the chapter is its several substantial case studies. In particular, a lengthy case study implements a point, circle and cylinder class hierarchy. The exercises ask the student to compare the creation of new classes by inheritance vs. composition; to extend the inheritance hierarchies discussed in the chapter; to write an inheritance hierarchy for quadrilaterals, trapezoids, parallelograms, rectangles and squares and to create a more general shape hierarchy with two-dimensional shapes and three-dimensional shapes. The chapter explains polymorphic behavior. When many classes are related through inheritance to a common superclass, each subclass object may be treated as a superclass object. This enables programs to be written in a general manner independent of the specific types of the subclass objects. New kinds of objects can be handled by the same program, thus making systems more extensible. Polymorphism enables programs to eliminate complex `switch` logic in favor of simpler “straight-line” logic. A video game screen manager, for example, can send a “draw” message to every object in a linked list of objects to be drawn. Each object knows
how to draw itself. A new type of object can be added to the program without modifying
that program as long as that new object also knows how to draw itself. This style of pro-
gramming is typically used to implement today’s popular graphical user interfaces. The
chapter distinguishes between abstract classes (from which objects cannot be instanti-
ated) and concrete classes (from which objects can be instantiated). The chapter also intro-
duces interfaces—sets of methods that must be defined by any class that implements
the interface. Interfaces are Java’s replacement for the dangerous (albeit powerful) feature of
C++ called multiple inheritance.

Abstract classes are useful for providing a basic set of methods and default implementa-
tion to classes throughout the hierarchy. Interfaces are useful in many situations similar
to abstract classes; however, interfaces do not include any implementation—interfaces
have no method bodies and no instance variables. A feature of the chapter is its three major
polymorphism case studies—a payroll system, a shape hierarchy headed up by an
abstract class and a shape hierarchy headed up by an interface. The chapter exercises
ask the student to discuss a number of conceptual issues and approaches, work with
abstract classes, develop a basic graphics package, modify the chapter’s employee
class—and pursue all these projects with polymorphic programming.

Chapter 10—Strings and Characters—deals with processing words, sentences,
characters and groups of characters. The key difference between Java and C here is that
Java strings are objects. This makes string manipulation more convenient and much safer
than in C where string and array manipulations are based on dangerous pointers. We
present classes String, StringBuffer, Character and StringTokenizer. For
each, we provide extensive live-code examples demonstrating most of their methods “in
action.” In all cases, we show output windows so that the reader can see the precise effects
of each of the string and character manipulations. Students will enjoy the card shuffling
and dealing example (which they will enhance in the exercises to the later chapters on graphics
and multimedia). A key feature of the chapter is an extensive collection of challenging
string-manipulation exercises related to limericks, pig Latin, text analysis, word pro-
cessing, printing dates in various formats, check protection, writing the word equivalent of
a check amount, Morse Code and metric-to-English conversions. Students will enjoy the
challenges of developing their own spell checker and crossword-puzzle generator.

Advanced Topics
Chapters 11, 12 and 13 were coauthored with our colleague, Mr. Tem Nieto of Deitel &
Associates, Inc. Tem’s infinite patience, attention to detail, illustration skills and creativity
are apparent throughout these chapters. [Take a fast peek at Figure 12.19 to see what hap-
pens when we turn Tem loose!]

Chapter 11—Graphics and Java2D—is the first of several chapters that present the
multimedia “sizzle” of Java. We consider Chapters 11 through 22 to be the book’s
advanced material. This is “fun stuff.” Traditional C and C++ programming are pretty
much confined to character-mode input/output. Some versions of C++ are supported by
platform-dependent class libraries that can do graphics, but using these libraries makes
your applications nonportable. Java’s graphics capabilities are platform independent and
hence, portable—and we mean portable in a worldwide sense. You can develop graphics-
intensive Java applets and distribute them over the World Wide Web to colleagues every-
where, and they will run nicely on the local Java platforms. We discuss graphics contexts
and graphics objects; drawing strings, characters and bytes; color and font control; screen manipulation and paint modes and drawing lines, rectangles, rounded rectangles, three-dimensional rectangles, ovals, arcs and polygons. We introduce the Java2D API, which provides powerful graphical manipulation tools. Figure 11.22 is an example of how easy it is to use the Java2D API to create complex graphics effects such as textures and gradients. The chapter has 23 figures that painstakingly illustrate each of these graphics capabilities with live-code™ examples, appealing screen outputs, detailed features tables and detailed line art. Some of the 27 exercises challenge students to develop graphical versions of their solutions to previous exercises on Turtle Graphics, the Knight’s Tour, the Tortoise and the Hare simulation, Maze Traversal and the Bucket Sort. Our companion book, Advanced Java 2 Platform How to Program, presents the Java 3D API.

Chapter 12—Graphical User Interface Components: Part 1—introduces the creation of applets and applications with user-friendly graphical user interfaces (GUIs). This chapter focuses on Java’s Swing GUI components. These platform-independent GUI components are written entirely in Java. This provides Swing GUI components with great flexibility—the GUI components can be customized to look like the computer platform on which the program executes, or they can use the standard Java look-and-feel that provides an identical user interface across all computer platforms. GUI development is a huge topic, so we divided it into two chapters. These chapters cover the material in sufficient depth to enable you to build “industrial-strength” GUI interfaces. We discuss the javax.swing package, which provides much more powerful GUI components than the java.awt components that originated in Java 1.0. Through its 16 programs and many tables and line drawings, the chapter illustrates GUI principles, the javax.swing hierarchy, labels, push buttons, lists, text fields, combo boxes, checkboxes, radio buttons, panels, handling mouse events, handling keyboard events and using three of Java’s simpler GUI layout managers, namely, FlowLayout, BorderLayout and GridLayout. The chapter concentrates on the delegation event model for GUI processing. The 33 exercises challenge the student to create specific GUIs, exercise various GUI features, develop drawing programs that let the user draw with the mouse and control fonts.

Chapter 13—Graphical User Interface Components: Part 2—continues the detailed Swing discussion started in Chapter 12. Through its 13 programs, as well as tables and line drawings, the chapter illustrates GUI design principles, the javax.swing hierarchy, text areas, subclassing Swing components, sliders, windows, menus, pop-up menus, changing the look-and-feel, and using three of Java’s advanced GUI layout managers, namely, BoxLayout, CardLayout and GridBagLayout. Two of the most important examples introduced in this chapter are a program that can run as either an applet or application and a program that demonstrates how to create a multiple document interface (MDI) graphical user interface. MDI is a complex graphical user interface in which one window—called the parent—acts as the controlling window for the application. This parent window contains one or more child windows—which are always graphically displayed within the parent window. Most word processors use MDI graphical user interfaces. The chapter concludes with a series of exercises that encourage the reader to develop substantial GUIs with the techniques and components presented in the chapter. One of the most challenging exercises in this chapter is a complete drawing application that asks the reader to create an object oriented-program that keeps track of the shapes the user has drawn. Other exercises use inheritance to subclass Swing compo-
nents and reinforce layout manager concepts. The first six chapters of our companion book, Advanced Java 2 Platform How to Program, are designed for courses in advanced GUI programming.

Chapter 14—Exception Handling—is one of the most important chapters in the book from the standpoint of building so-called “mission-critical” or “business-critical” applications that require high degrees of robustness and fault tolerance. Things do go wrong, and at today’s computer speeds—commonly hundreds of millions operations per second (with recent personal computers running at a billion or more instructions per second)—if they can go wrong they will, and rather quickly at that. Programmers are often a bit naive about using components. They ask, “How do I request that a component do something for me?” They also ask “What value(s) does that component return to me to indicate it has performed the job I asked it to do?” But programmers also need to be concerned with, “What happens when the component I call on to do a job experiences difficulty? How will that component signal that it had a problem?” In Java, when a component (e.g., a class object) encounters difficulty, it can “throw an exception.” The environment of that component is programmed to “catch” that exception and deal with it. Java’s exception-handling capabilities are geared to an object-oriented world in which programmers construct systems largely from reusable, prefabricated components built by other programmers. To use a Java component, you need to know not only how that component behaves when “things go well,” but also what exceptions that component throws when “things go poorly.” The chapter distinguishes between rather serious system Errors (normally beyond the control of most programs) and Exceptions (which programs generally deal with to ensure robust operation). The chapter discusses the vocabulary of exception handling. The try block executes program code that either executes properly or throws an exception if something goes wrong. Associated with each try block are one or more catch blocks that handle thrown exceptions in an attempt to restore order and keep systems “up and running” rather than letting them “crash.” Even if order cannot be fully restored, the catch blocks can perform operations that enable a system to continue executing, albeit at reduced levels of performance—such activity is often referred to as “graceful degradation.” Regardless of whether exceptions are thrown, a finally block accompanying a try block will always execute; the finally block normally performs cleanup operations like closing files and releasing resources acquired in the try block. The material in this chapter is crucial to many of the live-code examples in the remainder of the book. The chapter enumerates many of the Errors and Exceptions of the Java packages. The chapter has some of the most appropriate quotes in the book, thanks to Barbara Deitel’s painstaking research. The vast majority of the book’s Testing and Debugging Tips emerged naturally from the material in Chapter 14.

Chapter 15—Multithreading—deals with programming applets and applications that can perform multiple activities in parallel. Although our bodies are quite good at this (breathing, eating, blood circulation, vision, hearing, etc. can all occur in parallel), our conscious minds have trouble with this. Computers used to be built with a single, rather expensive processor. Today, processors are becoming so inexpensive that it is possible to build computers with many processors that work in parallel—such computers are called multiprocessors. The trend is clearly towards computers that can perform many tasks in parallel. Most of today’s programming languages, including C and C++, do not include built-in features for expressing parallel operations. These languages are often referred to as “sequen-
tial” programming languages or “single-thread-of-control” languages. Java includes capabilities to enable multithreaded applications (i.e., applications that can specify that multiple activities are to occur in parallel). This makes Java better prepared to deal with the more sophisticated multimedia, network-based multiprocessor-based applications programmers will develop. As we will see, multithreading is effective even on single-processor systems. For years, the “old guy” taught operating systems courses and wrote operating systems textbooks, but he never had a multithreaded language like Java available to demonstrate the concepts. In this chapter, we thoroughly enjoyed presenting multithreaded programs that demonstrate clearly the kinds of problems that can occur in parallel programming. There are all kinds of subtleties that develop in parallel programs that you simply never think about when writing sequential programs. A feature of the chapter is the extensive set of examples that show these problems and how to solve them. Another feature is the implementation of the “circular buffer,” a popular means of coordinating control between asynchronous, concurrent “producer” and “consumer” processes that, if left to run without synchronization, would cause data to be lost or duplicated incorrectly, often with devastating results. We discuss the monitor construct developed by C. A. R. Hoare and implemented in Java; this is a standard topic in operating systems courses. The chapter discusses threads and thread methods. It walks through the various thread states and state transitions with a detailed line drawing showing the life-cycle of a thread. We discuss thread priorities and thread scheduling and use a line drawing to show Java’s fixed-priority scheduling mechanism. We examine a producer/consumer relationship without synchronization, observe the problems that occur and solve the problem with thread synchronization. We implement a producer/consumer relationship with a circular buffer and proper synchronization with a monitor. We discuss daemon threads that “hang around” and perform tasks (e.g., “garbage collection”) when processor time is available. We discuss interface Runnable which enables objects to run as threads without having to subclass class Thread. We close with a discussion of thread groups which, for example, enable separation to be enforced between system threads like the garbage collector and user threads. The chapter has a nice complement of exercises. The featured exercise is the classic readers and writers problem, a favorite in upper level operating systems courses; citations appear in the exercises for students who wish to research this topic. This is an important problem in database-oriented transaction-processing systems. It raises subtle issues of solving problems in concurrency control while ensuring that every separate activity that needs to receive service does so without the possibility of “indefinite postponement,” that could cause some activities never to receive service—a condition also referred to as “starvation.” Operating systems professors will enjoy the projects implemented by Java-literate students. We can expect substantial progress in the field of parallel programming as Java’s multithreading capabilities enable large numbers of computing students to pursue parallel-programming class projects. As these students enter industry over the next several years, we expect a surge in parallel systems programming and parallel applications programming. We have been predicting this for decades—Java is making it a reality.

If this is your first Java book and you are an experienced computing professional, you may well be thinking, “Hey, this just keeps getting better and better. I can’t wait to get started programming in this language. It will let me do all kinds of stuff I would like to do, but that was never easy for me to do with the other languages I have used.” You’ve got it right. Java is an enabler. So, if you liked the multithreading discussion, hold onto your hat,
because Java will let you program multimedia applications and make them available instantaneously over the World Wide Web.

**Chapter 16—Files and Streams**—deals with input/output that is accomplished through streams of data directed to and from files. This is one of the most important chapters for programmers who will be developing commercial applications. Modern business is centered around data. In this chapter, we translate data (objects) into a persistent format usable by other applications. Being able to store data in files or move it across networks (Chapter 17) makes it possible for programs to save data and to communicate with each other. This is the real strength of software today. The chapter begins with an introduction to the data hierarchy from bits, to bytes, to fields, to records, to files. Next, Java’s simple view of files and streams is presented. We then present a walkthrough of the dozens of classes in Java’s extensive input/output files and streams class hierarchy. We put many of these classes to work in live-code examples in this chapter and in Chapter 17. We show how programs pass data to secondary storage devices, like disks, and how programs retrieve data already stored on those devices. Sequential-access files are discussed using a series of three programs that show how to open and close files, how to store data sequentially in a file and how to read data sequentially from a file. Random-access files are discussed using a series of four programs that show how to create a file sequentially for random access, how to read and write data to a file with random access and how to read data sequentially from a randomly accessed file. The fourth random-access program combines many of the techniques of accessing files both sequentially and randomly into a complete transaction-processing program. We discuss buffering and how it helps programs that do significant amounts of input/output perform better. We discuss class *File* which programs use to obtain a variety of information about files and directories. We explain how objects can be output to, and input from, secondary storage devices. Students in our industry seminars have told us that, after studying the material on file processing, they were able to produce substantial file-processing programs that were immediately useful to their organizations. The exercises ask the student to implement a variety of programs that build and process sequential-access files and random-access files.

**Chapter 17—Networking**—deals with applets and applications that can communicate over computer networks. This chapter presents Java’s lowest level networking capabilities. We write programs that “walk the Web.” The chapter examples illustrate an applet interacting with the browser in which it executes, creating a mini Web browser, communicating between two Java programs using streams-based sockets and communicating between two Java programs using packets of data. A key feature of the chapter is the live-code implementation of a collaborative client/server Tic-Tac-Toe game in which two clients play Tic-Tac-Toe with one another arbitrated by a multithreaded server—great stuff! The multithreaded server architecture is exactly what is used today in popular UNIX and Windows NT network servers. The capstone example in the chapter is the Deitel Messenger case study, which simulates many of today’s popular instant-messaging applications that enable computers users to communicate with friends, relatives and coworkers over the Internet. This 1130-line, multithreaded, client/server case study uses most of the programming techniques presented up to this point in the book. The messenger application also introduces multicasting, which enables a program to send packets of data to groups of clients. The chapter has a nice collection of exercises including several suggested modifications to the multithreaded server example. Our companion book, *Advanced Java 2 Platform*
How to Program, offers a much deeper treatment of networking and distributed computing, with topics including remote method invocation (RMI), servlets, JavaServer Pages (JSP), Java 2 Enterprise Edition, wireless Java (and the Java 2 Micro Edition) and Common Object Request Broker Architecture (CORBA).

Chapter 18—Multimedia: Images, Animation and Audio—is the first of two chapters that present Java’s capabilities for making computer applications come alive (Chapter 22 offers an extensive treatment of the Java Media Framework). It is remarkable that students in first programming courses will be writing applications with all these capabilities. The possibilities are intriguing. Imagine having access (over the Internet and through CD-ROM technology) to vast libraries of graphics images, audios and videos and being able to weave your own together with those in the libraries to form creative applications. Already, most new computers sold come “multimedia equipped.” Students can create extraordinary term papers and classroom presentations with components drawn from vast public-domain libraries of images, line drawings, voices, pictures, videos, animations and the like. A “paper” when many of us were in the earlier grades was a collection of characters, possibly handwritten, possibly typewritten. A “paper” today can be a multimedia “extravaganza” that makes the subject matter come alive. It can hold your interest, pique your curiosity and make you feel what the subjects of the paper felt when they were making history. Multimedia is making science labs much more exciting. Textbooks are coming alive. Instead of looking at a static picture of some phenomenon, you can watch that phenomenon occur in a colorful, animated, presentation with sounds, videos and various other effects, leveraging the learning process. People are able to learn more, learn it in more depth and experience more viewpoints.

The chapter discusses images and image manipulation, audios and animation. A feature of the chapter is the image maps that enable a program to sense the presence of the mouse pointer over a region of an image, without clicking the mouse. We present a live-code image-map application with the icons from the programming tips you have seen in this chapter and will see throughout the book. As the user moves the mouse pointer across the seven icon images, the type of tip is displayed, either “Good Programming Practice” for the thumbs-up icon, “Portability Tip” for the bug with the suitcase icon and so on. Once you have read the chapter, you will be eager to try out all these techniques, so we have included 35 problems to challenge and entertain you (more are provided in Chapter 22). Here are some of the exercises that you may want to turn into term projects:

15 Puzzle
Analog Clock
Animation
Artist
Calendar/Tickler File
Calling Attention to an Image
Coloring Black and White Images
Crossword
Fireworks Designer
Game of Pool
Horse Race
Image Flasher
Image Zooming
Jigsaw Puzzle Generator
Kaleidoscope
Limericks
Maze Generator and Walker
Multimedia Simpletron Simulator
One-Armed Bandit
Random Inter-Image Transition
Randomly Erasing an Image
Reaction Time Tester
Rotating Images
Scrolling Image Marquee
Scrolling Text Marquee
Shuffleboard
Text Flasher

You are going to have a great time attacking these problems! Some will take a few hours and some are great term projects. We see all kinds of opportunities for multimedia electives starting to appear in the university computing curriculum. We hope you will have contests with your classmates to develop the best solutions to several of these problems.
Chapter 19—Data Structures—is particularly valuable in second- and third-level university courses. The chapter discusses the techniques used to create and manipulate dynamic data structures, such as linked lists, stacks, queues (i.e., waiting lines) and trees. The chapter begins with discussions of self-referential classes and dynamic memory allocation. We proceed with a discussion of how to create and maintain various dynamic data structures. For each type of data structure, we present live-code programs and show sample outputs. Although it is valuable to know how these classes are implemented, Java programmers will quickly discover that many of the data structures they need are already available in class libraries, such as Java’s own java.util that we discuss in Chapter 20 and Java Collections that we discuss in Chapter 21. The chapter helps the student master Java-style references (i.e., Java’s replacement for the more dangerous pointers of C and C++). One problem when working with references is that students could have trouble visualizing the data structures and how their nodes are linked together. So we present illustrations that show the links and the sequence in which they are created. The binary tree example is a nice capstone for the study of references and dynamic data structures. This example creates a binary tree; enforces duplicate elimination and introduces recursive preorder, inorder and postorder tree traversals. Students have a genuine sense of accomplishment when they study and implement this example. They particularly appreciate seeing that the inorder traversal prints the node values in sorted order. The chapter includes a substantial collection of exercises. A highlight of the exercises is the special section “Building Your Own Compiler.” This exercise is based on earlier exercises that walk the student through the development of an infix-to-postfix conversion program and a postfix-expression evaluation program. We then modify the postfix evaluation algorithm to generate machine-language code. The compiler places this code in a file (using techniques the student mastered in Chapter 16). Students then run the machine language produced by their compilers on the software simulators they built in the exercises of Chapter 7! The many exercises include a supermarket simulation using queueing, recursively searching a list, recursively printing a list backwards, binary tree node deletion, level-order traversal of a binary tree, printing trees, writing a portion of an optimizing compiler, writing an interpreter, inserting/deleting anywhere in a linked list, analyzing the performance of binary tree searching and sorting and implementing an indexed list class.

Chapter 20—Java Utilities Package and Bit Manipulation—walks through the classes of the java.util package and discusses each of Java’s bitwise operators. This is a nice chapter for reinforcing the notion of reuse. When classes already exist, it is much faster to develop software by simply reusing these classes than by “reinventing the wheel.” Classes are included in class libraries because the classes are generally useful, correct, performance tuned, portability certified and/or for a variety of other reasons. Someone has invested considerable work in preparing these classes so why should you write your own? The world’s class libraries are growing at a phenomenal rate. Given this, your skill and value as a programmer will depend on your familiarity with what classes exist and how you can reuse them cleverly to develop high-quality software rapidly. University data structures courses will be changing drastically over the next several years because most important data structures are already implemented in widely available class libraries. This chapter discusses many classes. Two of the most useful are Vector (a dynamic array that can grow and shrink as necessary) and Stack (a dynamic data structure that allows insertions and deletions from only one end—called the top—thus ensuring last-in-first-out behavior).
beauty of studying these two classes is that they are related through inheritance, as is dis-
cussed in Chapter 9, so the java.util package itself implements some classes in terms
of others, thus avoiding reinventing the wheel and taking advantage of reuse. We also dis-

cuss classes Dictionary, Hashtable, Properties (for creating and manipulating persistent
Hashtables), Random and BitSet. The discussion of BitSet includes live
code for one of the classic applications of BitSets, namely the Sieve of Eratosthenes,
used for determining prime numbers. The chapter discusses in detail Java’s powerful bit-
manipulation capabilities, which enable programmers to exercise lower level hardware
capabilities. This helps programs process bit strings, set individual bits on or off and store
information more compactly. Such capabilities—inherted from C—are characteristic of
low-level assembly languages and are valued by programmers writing system software
such as operating systems and networking software.

Chapter 21—Collections—discusses many of the Java 2 classes (of the
java.util package) that provide predefined implementations of many of the data
structures discussed in Chapter 19. This chapter, too, reinforces the notion of reuse.
These classes are modeled after a similar class library in C++—the Standard Template
Library. Collections provide Java programmers with a standard set of data structures for
storing and retrieving data and a standard set of algorithms (i.e., procedures) that allow
programmers to manipulate the data (such as searching for particular data items and
sorting data into ascending or descending order). The chapter examples demonstrate col-
clections, such as linked lists, trees, maps and sets, and algorithms for searching, sorting,
finding the maximum value, finding the minimum value and so on. Each example clearly
shows how powerful and easy to use collections are. The exercises suggest modifications
to the chapter examples and ask the reader to reimplement data structures presented in
Chapter 19 using collections.

Chapter 22—Java Media Framework and Java Sound—is the second of our two
chapters dedicated to Java’s tremendous multimedia capabilities. This chapter focusses on
the Java Media Framework (JMF) and the Java Sound API. The Java Media Framework
provides both audio and video capabilities. With the JMF, a Java program can play audio
and video media and capture audio and video media from devices such as microphones and
video cameras. Many of today’s multimedia applications involve sending audio or video
feeds across the Internet. For example, you can visit the cnn.com Web site to watch or
listen to live news conferences, and many people listen to Internet-based radio stations
through their Web browsers. The JMF enables Java developers to create so-called
streaming media applications, in which a Java program sends live or recorded audio or
video feeds across the Internet to other computers, then applications on those other com-
puters play the media as it arrives over the network. The JavaSound APIs enable programs
to manipulate Musical Instrument Digital Interface (MIDI) sounds and captured media
(i.e., media from a device such as a microphone). This chapter concludes with a substantial
MIDI-processing application that enables users to select MIDI files to play and record new
MIDI files. Users can create their own MIDI music by interacting with the application’s
simulated synthesizer keyboard. In addition, the application can synchronize playing the
notes in a MIDI file with pressing the keys on the simulated synthesizer keyboard—similar
to a player piano! As with Chapter 18, once you read this chapter, you will be eager to try
all these techniques, so we have included 44 additional multimedia exercises to challenge
and entertain you. Some of the interesting projects include the following:
Appendix A—Java Demos—presents a huge collection of some of the best Java demos available on the Web. Many of these sites make their source code available to you, so you can download the code and add your own features—a truly great way to learn Java! We encourage our students to do this, and we’re amazed at the results! You should start your search by checking out the Sun Microsystems applet Web page, java.sun.com/applets. You can save time finding the best demos by checking out JARS (the Java Applet Rating Service) at www.jars.com. Here’s a list of some of the demos mentioned in Appendix A (the URLs and descriptions of each are in Appendix A):

<table>
<thead>
<tr>
<th>Demo Name</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bouncing Ball Physics Demo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Craps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digital Clock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flight Simulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Karaoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kinetics Physics Demo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knight’s Tour Walker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Morse Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP3 Player</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multimedia Authoring System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pinball Machine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Roulette</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Story Teller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tic-Tac-Toe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tortoise and the Hare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Towers of Hanoi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video Conferencing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video Games</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Appendix B—Java Resources—presents some of the best Java resources available on the Web. This is a great way for you to get into the “world of Java.” The appendix lists various Java resources, such as consortia, journals and companies that make various key Java-related products. Here are some of the resources mentioned in Appendix B:

- animated applets
- applets/applications
- arts and entertainment
- audio sites
- books
- Borland JBuilder IDE
- conferences
- consultants
- contests
- CORBA homepage
- current information
- databases
- demos (many with source code)
- developer’s kit
- development tools
- discussion groups
- documentation
- downloadable applets
- FAQs (frequently asked ?s)
- games
- graphics
- IBM Developers Java Zone
- Intelligence.com
- Java Applet Rating Service
- Java Developer Connection
- Java Developer’s Journal
- Java Media Framework
- Java Report
- Java tools
- Java Toys
- Java Users Group (JUGs)
- Java Woman
- JavaWorld on-line magazine
- learning Java
- links to Java sites
- lists of resources
- lists of what is new and cool
- live chat sessions on Java
- multimedia collections
- NASA multimedia gallery
- NetBeans IDE
- news
- news:comp.lang.java
- Object Management Group
- products
- projects
- publications
- puzzles
- reference materials
- resources
- seminars
- sites
- software
- Sun Microsystems
- SunWorld on-line magazine
- Team Java
- The Java Tutorial
- trade shows
- training (please call us!)
- tutorials for learning java
- URLs for Java applets
- www.javaworld.com
- Yahoo (Web search engine)
Appendix C—Operator Precedence Chart—lists each of the Java operators and indicates their relative precedence and associativity. We list each operator on a separate line and include the full name of the operator.

Appendix D—ASCII Character Set—lists the characters of the ASCII (American Standard Code for Information Interchange) character set and indicates the character code value for each. Java uses the Unicode character set with 16-bit characters for representing all of the characters in the world’s “commercially significant” languages. Unicode includes ASCII as a subset. Currently, most English-speaking countries are using ASCII and just beginning to experiment with Unicode.

Appendix E—Number Systems—discusses the binary (base 2), decimal (base 10), octal (base 8) and hexadecimal (base 16) number systems. This material is valuable for introductory courses in computer science and computer engineering. The appendix is presented with the same pedagogic learning aids as the chapters of the book. A nice feature of the appendix is its 31 exercises, 19 of which are self-review exercises with answers.

Appendix F—Creating javadoc Documentation—introduces the javadoc documentation-generation tool. Sun Microsystems uses javadoc to document the Java APIs. The example in this appendix takes the reader through the javadoc documentation process. First, we introduce the comment style and tags that javadoc recognizes and uses to create documentation. Next, we discuss the commands and options used to run the utility. Finally, we examine the source files javadoc uses and the HTML files javadoc creates.

1.18 (Optional) A Tour of the Case Study on Object-Oriented Design with the UML

In this and the next section, we tour the two optional major features of the book—the optional case study of object-oriented design with the UML and our introduction to design patterns. The case study involving object-oriented design with the UML is an important addition to Java How to Program, Fourth Edition. This tour previews the contents of the “Thinking About Objects” sections and discusses how they relate to the case study. After completing this case study, you will have completed an object-oriented design and implementation for a significant Java application.

Section 1.15—Thinking About Objects: Introduction to Object Technology and the Unified Modeling Language

This section introduces the object-oriented design case study with the UML. We provide a general background of what objects are and how they interact with other objects. We also discuss briefly the state of the software-engineering industry and how the UML has influenced object-oriented analysis and design processes.

Section 2.9—(Optional Case Study) Thinking About Objects: Examining the Problem Statement

Our case study begins with a problem statement that specifies the requirements for a system that we will create. In this case study, we design and implement a simulation of an elevator system in a two-story building. The application user can “create” a person on either floor. This person then walks across the floor to the elevator, presses a button, waits for the elevator to arrive and rides it to the other floor. We provide the design of our elevator system after investigating the structure and behavior of object-oriented systems in general. We dis-
discuss how the UML will facilitate the design process in subsequent “Thinking About Object” sections by providing us with several types of diagrams to model our system. Finally, we provide a list of URL and book references on object-oriented design with the UML. You might find these references helpful as you proceed through our case-study presentation.

Section 3.8—(Optional Case Study) Thinking About Objects: Identifying the Classes in the Problem Statement
In this section, we design the elevator-simulation model, which represents the operations of the elevator system. We identify the classes, or “building blocks,” of our model by extracting the nouns and noun phrases from the problem statement. We arrange these classes into a UML class diagram that describes the class structure of our model. The class diagram also describes relationships, known as associations, among classes (for example, a person has an association with the elevator, because the person rides the elevator). Lastly, we extract from the class diagram another type of diagram in the UML—the object diagram. The object diagram models the objects (instances of classes) at a specific time in our simulation.

Section 4.14—(Optional Case Study) Thinking About Objects: Identifying Class Attributes
A class contains both attributes (data) and operations (behaviors). This section focuses on the attributes of the classes discussed in Section 3.7. As we see in later sections, changes in an object’s attributes often affect the behavior of that object. To determine the attributes for the classes in our case study, we extract the adjectives describing the nouns and noun phrases (which defined our classes) from the problem statement, then place the attributes in the class diagram we create in Section 3.7.

Section 5.11—(Optional Case Study) Thinking About Objects: Identifying Objects’ States and Activities
An object, at any given time, occupies a specific condition called a state. A state transition occurs when that object receives a message to change state. The UML provides the state-chart diagram, which identifies the set of possible states that an object may occupy and models that object’s state transitions. An object also has an activity—the work performed by an object in its lifetime. The UML provides the activity diagram—a flowchart that models an object’s activity. In this section, we use both types of diagrams to begin modeling specific behavioral aspects of our elevator simulation, such as how a person rides the elevator and how the elevator responds when a button is pressed on a given floor.

Section 6.16—(Optional Case Study) Thinking About Objects: Identifying Class Operations
In this section, we identify the operations, or services, of our classes. We extract from the problem statement the verbs and verb phrases that specify the operations for each class. We then modify the class diagram of Fig. 3.16 to include each operation with its associated class. At this point in the case study, we will have gathered all information possible from the problem statement. However, as future chapters introduce such topics as inheritance, event-handling and multithreading, we will modify our classes and diagrams.

Section 7.10—(Optional Case Study) Thinking About Objects: Collaboration Among Objects
At this point, we have created a “rough sketch” of the model for our elevator system. In this section, we see how it works. We investigate the behavior of the model by discussing col-
laborations—messages that objects send to each other to communicate. The class opera-
tions that we discovered in Section 6.16 turn out to be the collaborations among the objects
in our system. We determine the collaborations in our system, then collect them into a col-
laboration diagram—the UML diagram for modeling collaborations. This diagram reveals
which objects collaborate and when. We present a collaboration diagram of the people en-
tering and exiting the elevator.

Section 8.17—(Optional Case Study) Thinking About Objects: Starting to Program the
Classes for the Elevator Simulation
In this section, we take a break from designing the behavior of our system. We begin the
implementation process to emphasize the material discussed in Chapter 8. Using the UML
class diagram of Section 3.7 and the attributes and operations discussed in Sections 4.14
and 6.16, we show how to implement a class in Java from a design. We do not implement
all classes—because we have not completed the design process. Working from our UML
diagrams, we create code for the Elevator class.

Section 9.23—(Optional Case Study) Thinking About Objects: Incorporating
Inheritance into the Elevator Simulation
Chapter 9 begins our discussion of object-oriented programming. We consider inherit-
ance—classes sharing similar characteristics may inherit attributes and operations from a
“base” class. In this section, we investigate how our elevator simulation can benefit from
using inheritance. We document our discoveries in a class diagram that models inheritance
relationships—the UML refers to these relationships as generalizations. We modify the
class diagram of Section 3.7 by using inheritance to group classes with similar characteris-
tics. We continue implementing the Elevator class of Section 8.17 by incorporating in-
heritance.

Section 10.22—(Optional Case Study) Thinking About Objects: Event Handling
In this section, we include interfaces necessary for the objects in our elevator simulation to
send messages to other objects. In Java, objects often communicate by sending an event—a
notification that some action has occurred. The object receiving the event then performs
an action in response to the type of event received—this is known as event handling. In
Section 7.10, we outlined the message passing, or the collaborations, in our model, using a
collaboration diagram. We now modify this diagram to include event handling, and, as an
example, we explain in detail how doors in our simulation open upon the elevator’s arrival.

Section 11.10—(Optional Case Study) Thinking About Objects: Designing Interfaces
with the UML
In this section, we design a class diagram that models the relationships between classes and
interfaces in our simulation—the UML refers to these relationships as realizations. In ad-
dition, we list all operations that each interface provides to the classes. Lastly, we show how
to create the Java classes that implement these interfaces. As in Section 8.17 and
Section 9.23, we use class Elevator to demonstrate the implementation.

Section 12.16 - (Optional Case Study) Thinking About Objects: Use Cases
Chapter 12 discusses user interfaces that enable a user to interact with a program. In this
section, we discuss the interaction between our elevator simulation and its user. Specifical-
ly, we investigate the scenarios that may occur between the application user and the simu-
lation itself—this set of scenarios is called a use case. We model these interactions, using use-case diagrams of the UML. We then discuss the graphical user interface for our simulation, using our use-case diagrams.

Section 13.17—(Optional Case Study) Thinking About Objects: Model-View-Controller
We designed our system to consist of three components, each having a distinct responsibility. By this point in the case study, we have almost completed the first component, called the model, which contains data that represent the simulation. We design the view—the second component, dealing with how the model is displayed—in Section 22.8. We design the controller—the component that allows the user to control the model—in Section 12.16. A system such as ours that uses the model, view and controller components is said to adhere to Model-View-Controller (MVC) architecture. In this section, we explain the advantages of using this architecture to design software. We use the UML component diagram to model the three components, then implement this diagram as Java code.

Section 15.12—(Optional Case Study) Thinking About Objects: Multithreading
In the real world, objects operate and interact concurrently. Java is a multithreaded language, which enables the objects in our simulation to act seemingly independently from each other. In this section, we declare certain objects as “threads” to enable these objects to operate concurrently. We modify the collaboration diagram originally presented in Section 7.10 (and modified in Section 10.22) to incorporate multithreading. We present the UML sequence diagram for modeling interactions in a system. This diagram emphasizes the chronological ordering of messages. We use a sequence diagram to model how a person inside the simulation interacts with the elevator. This section concludes the design of the model portion of our simulation. We design how this model is displayed in Section 22.9, then implement this model as Java code in Appendix H.

Section 22.9—(Optional Case Study) Thinking About Objects: Animation and Sound in the View
This section designs the view, which specifies how the model portion of the simulation is displayed. Chapter 18 presents several techniques for integrating animation in programs, and Chapter 22 presents techniques for integrating sound. Section 22.9 uses some of these techniques to incorporate sound and animation into our elevator simulation. Specifically, this section deals with animating the movements of people and our elevator, generating sound effects and playing “elevator music” when a person rides the elevator. This section concludes the design of our elevator simulation. Appendices G, H and I implement this design as a 3,594-line, fully operational Java program.

Appendix G—Elevator Events and Listener Interfaces
[Note: This appendix is on the CD that accompanies this book.] As we discussed in Section 10.22, several objects in our simulation interact with each other by sending messages, called events, to other objects wishing to receive these events. The objects receiving the events are called listener objects—these must implement listener interfaces. In this section, we implement all event classes and listener interfaces used by the objects in our simulation.

Appendix H—Elevator Model
[Note: This appendix is on the CD that accompanies this book.] The majority of the case study involved designing the model (i.e., the data and logic) of the elevator simulation. In
this section, we implement that model in Java. Using all the UML diagrams we created, we present the Java classes necessary to implement the model. We apply the concepts of object-oriented design with the UML and object-oriented programming and Java that you learned in the chapters.

Appendix I—Elevator View
[Note: This appendix is on the CD that accompanies this book.] The final section implements how we display the model from Appendix H. We use the same approach to implement the view as we used to implement the model—we create all the classes required to run the view, using the UML diagrams and key concepts discussed in the chapters. By the end of this section, you will have completed an “industrial-strength” design and implementation of a large-scale system. You should feel confident tackling larger systems, such as the 8000-line Enterprise Java case study we present in our companion book Advanced Java 2 Platform How to Program and the kinds of applications that professional software engineers build. Hopefully, you will move on to even deeper study of object-oriented design with the UML.

1.19 (Optional) A Tour of the “Discovering Design Patterns” Sections
Our treatment of design patterns is spread over five optional sections of the book. We overview those sections here.

Section 9.24—(Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns
This section provides tables that list the sections in which we discuss the various design patterns. We divide the discussion of each section into creational, structural and behavioral design patterns. Creational patterns provide ways to instantiate objects, structural patterns deal with organizing objects and behavioral patterns deal with interactions between objects. The remainder of the section introduces some of these design patterns, such as the Singleton, Proxy, Memento and State design patterns. Finally, we provide several URLs for further study on design patterns.

Section 13.18—(Optional) Discovering Design Patterns: Design Patterns Used in Packages java.awt and javax.swing
This section contains most of our design-patterns discussion. Using the material on Java Swing GUI components in Chapters 12 and 13, we investigate some examples of pattern use in packages java.awt and javax.swing. We discuss how these classes use the Factory Method, Adapter, Bridge, Composite, Chain-of-Responsibility, Command, Observer, Strategy and Template Method design patterns. We motivate each pattern and present examples of how to apply them.

Section 15.13—(Optional) Discovering Design Patterns: Concurrent Design Patterns
Developers have introduced several design patterns since those described by the gang of four. In this section, we discuss concurrency design patterns, including Single-Threaded Execution, Guarded Suspension, Balking, Read/Write Lock and Two-Phase Termination—these solve various design problems in multithreaded systems. We investigate how class java.lang.Thread uses concurrency patterns.

© Copyright 1992-2002 by Deitel & Associates, Inc. All Rights Reserved. 7/8/01
Section 17.11—(Optional) Discovering Design Patterns: Design Patterns Used in Packages java.io and java.net

Using the material on files, streams and networking in Chapters 16 and 17, we investigate some examples of pattern use in packages java.io and java.net. We discuss how these classes use the Abstract Factory, Decorator and Facade design patterns. We also consider architectural patterns, which specify a set of subsystems—aggregates of objects that each collectively comprise a major system responsibility—and how these subsystems interact with each other. We discuss the popular Model-View-Controller and Layers architectural patterns.

Section 21.12—(Optional) Discovering Design Patterns: Design Patterns Used in Package java.util

Using the material on data structures and collections in Chapters 19, 20 and 21, we investigate pattern use in package java.util. We discuss how these classes use the Prototype and Iterator design patterns. This section concludes the discussion on design patterns. After finishing the Discovering Design Patterns material, you should be able to recognize and use key design patterns and have a better understanding of the workings of the Java API. After completing this material, we recommend that you move on to the gang-of-four book.

Well, there you have it! We have worked hard to create this book and its optional Cyber Classroom version. The book is loaded with live-code examples, programming tips, self-review exercises and answers, challenging exercises and projects, and numerous study aids to help you master the material. Java is a powerful programming language that will help you write programs quickly and effectively. And Java is a language that scales nicely into the realm of enterprise-systems development to help organizations build their key information systems. As you read the book, if something is not clear, or if you find an error, please write to us at deitel@deitel.com. We will respond promptly, and we will post corrections and clarifications on our Web site, www.deitel.com.

We hope you enjoy learning with Java How to Program: Fourth Edition as much as we enjoyed writing it!

SUMMARY

• Software controls computers (often referred to as hardware).

• Java is one of today’s most popular software-development languages.

• Java was developed by Sun Microsystems. Sun provides an implementation of the Java 2 Platform, Standard Edition called the Java 2 Software Development Kit (J2SDK), version 1.3.1 that includes the minimum set of tools you need to write software in Java.

• Java is a fully object-oriented language with strong support for proper software-engineering techniques.

• A computer is a device capable of performing computations and making logical decisions at speeds millions, even billions, of times faster than human beings can.

• Computers process data under the control of sets of instructions called computer programs. These computer programs guide the computer through orderly sets of actions specified by people called computer programmers.
• The various devices that comprise a computer system (such as the keyboard, screen, disks, memory and processing units) are referred to as hardware.

• The computer programs that run on a computer are referred to as software.

• The input unit is the “receiving” section of the computer. It obtains information (data and computer programs) from various input devices and places this information at the disposal of the other units so that the information may be processed.

• The output unit is the “shipping” section of the computer. It takes information processed by the computer and places it on output devices to make it available for use outside the computer.

• The memory unit is the rapid access, relatively low-capacity “warehouse” section of the computer. It retains information that has been entered through the input unit so that the information may be made immediately available for processing when it is needed and retains information that has already been processed until that information can be placed on output devices by the output unit.

• The arithmetic and logic unit (ALU) is the “manufacturing” section of the computer. It is responsible for performing calculations such as addition, subtraction, multiplication and division and for making decisions.

• The central processing unit (CPU) is the “administrative” section of the computer. It is the computer’s coordinator and is responsible for supervising the operation of the other sections.

• The secondary storage unit is the long-term, high-capacity “warehousing” section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (such as disks) until they are needed, possibly hours, days, months or even years later.

• Early computers were capable of performing only one job or task at a time. This form of computer operation often is called single-user batch processing.

• Software systems called operating systems were developed to help make it more convenient to use computers. Early operating systems managed the smooth transition between jobs and minimized the time it took for computer operators to switch between jobs.

• Multiprogramming involves the “simultaneous” operation of many jobs on the computer—the computer shares its resources among the jobs competing for its attention.

• Timesharing is a special case of multiprogramming in which dozens or even hundreds of users share a computer through terminals. The computer runs a small portion of one user’s job, then moves on to service the next user. The computer does this so quickly that it might provide service to each user several times per second, so programs appear to run simultaneously.

• An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results, as with previous modes of computing.

• In 1977, Apple Computer popularized the phenomenon of personal computing.

• In 1981, IBM introduced the IBM Personal Computer. Almost overnight, personal computing became legitimate in business, industry and government organizations.

• Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in local area networks (LANs) within an organization. This led to the phenomenon of distributed computing, in which an organization’s computing is distributed over networks to the sites at which the real work of the organization is performed.

• Today, information is shared easily across computer networks where some computers called file servers offer a common store of programs and data that may be used by client computers distributed throughout the network—hence the term client/server computing.

• Java has become the language of choice for developing Internet-based applications (and for many other purposes).
Computer languages may be divided into three general types: machine languages, assembly languages and high-level languages.

Any computer can directly understand only its own machine language. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent.

English-like abbreviations formed the basis of assembly languages. Translator programs called assemblers convert assembly-language programs to machine language at computer speeds.

Compilers translate high-level language programs into machine-language programs. High-level languages (like Java) contain English words and conventional mathematical notations.

Interpreter programs directly execute high-level language programs without the need for compiling those programs into machine language.

Although compiled programs execute much faster than interpreted programs, interpreters are popular in program-development environments in which programs are recompiled frequently as new features are added and errors are corrected.

Objects are essentially reusable software components that model items in the real world. Modular, object-oriented design and implementation approaches make software-development groups more productive than is possible with previous popular programming techniques such as structured programming. Object-oriented programs are often easier to understand, correct and modify.

Java originated at Sun Microsystems as a project for intelligent consumer-electronic devices.

When the World Wide Web exploded in popularity in 1993, Sun people saw the immediate potential of using Java to create Web pages with so-called dynamic content.

Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of Web servers, to provide applications for consumer devices and so on.

Java programs consist of pieces called classes. Classes consist of pieces called methods that perform tasks and return information when they complete their tasks.

Most Java programmers use rich collections of existing classes in Java class libraries.

FORTRAN (FORmula TRANslator) was developed by IBM Corporation between 1954 and 1957 for scientific and engineering applications that require complex mathematical computations.

COBOL (COmmon Business Oriented Language) was developed in 1959 by a group of computer manufacturers and government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data.

Pascal was designed at about the same time as C. It was created by Professor Nicklaus Wirth and was intended for academic use.

Basic was developed in 1965 at Dartmouth College as a simple language to help novices become comfortable with programming.

Structured programming is a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify.

The Ada language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. One important capability of Ada is called multitasking; this allows programmers to specify that many activities are to occur in parallel.

Most high-level languages—including C and C++—generally allow the programmer to write programs that perform only one activity at a time. Java, through a technique called multithreading, enables programmers to write programs with parallel activities.

The Internet was developed more than three decades ago with funding supplied by the Department of Defense. Originally designed to connect the main computer systems of about a dozen universi-
ties and research organizations, the Internet today is accessible by hundreds of millions of computers worldwide.

- The Web allows computer users to locate and view multimedia-intensive documents over the Internet.
- Java systems generally consist of several parts: an environment, the language, the Java Applications Programming Interface (API) and various class libraries.
- Java programs normally go through five phases to be executed—edit, compile, load, verify and execute.
- Java program file names end with the `.java` extension.

- The Java compiler (`javac`) translates a Java program into bytecodes—the language understood by the Java interpreter. If a program compiles correctly, the compiler produces a file with the `.class` extension. This is the file containing the bytecodes that are interpreted during the execution phase.
- A Java program must first be placed in memory before it can execute. This is done by the class loader, which takes the `.class` file (or files) containing the bytecodes and transfers it to memory. The `.class` file can be loaded from a disk on your system or over a network.
- An application is a program that is normally stored and executed on the user’s local computer.
- An applet is a small program that is normally stored on a remote computer that users connect to via a Web browser. Applets are loaded from a remote computer into the browser, executed in the browser and discarded when execution completes.
- Applications are loaded into memory, then executed by the `java` interpreter.
- Browsers are used to view HTML (Hypertext Markup Language) documents on the World Wide Web.

  - When the browser sees an applet in an HTML document, the browser launches the Java class loader to load the applet. The browsers that support Java each have a built-in Java interpreter. Once the applet is loaded, the Java interpreter in the browser begins executing the applet.
  - Applets can also be executed from the command line using the `appletviewer` command provided with the Java 2 Software Development Kit (J2SDK). The `appletviewer` is commonly referred to as the minimum browser—it knows only how to interpret applets.
  - Before the bytecodes in an applet are executed by the Java interpreter built into a browser or the `appletviewer`, they are verified by the bytecode verifier to ensure that the bytecodes for downloaded classes are valid and that they do not violate Java’s security restrictions.
  - An intermediate step between interpreters and compilers is a just-in-time (JIT) compiler that, as the interpreter runs, produces compiled code for the programs and executes the programs in machine language rather than reinterpreting them. JIT compilers do not produce machine language that is as efficient as a full compiler.
  - For organizations wanting to do heavy-duty information-systems development, Integrated Development Environments (IDEs) are available from the major software suppliers. The IDEs provide many tools for supporting the software-development process.
  - Object orientation is a natural way of thinking about the world and of writing computer programs.
  - The Unified Modeling Language (UML) is a graphical language that allows people who build systems to represent their object-oriented designs in a common notation.
  - Humans think in terms of objects. We possess the marvelous ability of abstraction, which enables us to view screen images as people, planes, trees and mountains rather than as individual dots of color (called pixels—for “picture elements”).
Humans learn about objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors.

Object-oriented design (OOD) models real-world objects. It takes advantage of class relationships, where objects of a certain class—such as a class of vehicles—have the same characteristics. It takes advantage of inheritance relationships, and even multiple-inheritance relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.

OOD encapsulates data (attributes) and functions (behavior) into objects; the data and functions of an object are intimately tied together.

Objects have the property of information hiding. This means that, although objects may know how to communicate with one another across well-defined interfaces, objects normally are not allowed to know how other objects are implemented.

Languages such as Java are object-oriented—programming in such a language is called object-oriented programming (OOP) and allows designers to implement the object-oriented design as a working system.

In Java, the unit of programming is the class from which objects are eventually instantiated (a fancy term for “created”). Java classes contain methods (which implement class behaviors) and attributes (which implement class data).

Java programmers concentrate on creating their own user-defined types, called classes. Each class contains data and the set of functions that manipulate that data. The data components of a Java class are called attributes. The function components of a Java class are called methods.

An instance of a user-defined type (i.e., a class) is called an object.

Classes can also have relationships with other classes. These relationships are called associations.

With object technology, we can build much of the software we will need by combining “standardized, interchangeable parts” called classes.

The process of analyzing and designing a system from an object-oriented point of view is called object-oriented analysis and design (OOAD).

The Unified Modeling Language (the UML) is now the most widely used graphical representation scheme for modeling object-oriented systems. Those who design systems use the language (in the form of graphical diagrams) to model their 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. Using design patterns can substantially reduce the complexity of the design process.

Design patterns benefit system developers by helping to construct reliable software using proven architectures and accumulated industry expertise, promoting design reuse in future systems, identifying common mistakes and pitfalls that occur when building systems, helping to design systems independently of the language in which they will be implemented, establishing a common design vocabulary among developers and shortening the design phase in a software-development process.

Designers use design patterns to construct sets of classes and objects.

Creational design patterns describe techniques to instantiate objects (or groups of objects).

Structural design patterns allow designers to organize classes and objects into larger structures.

Behavioral design patterns assign responsibilities to objects.

TERMINOLOGY

abstraction
Ada
ALU (arithmetic and logic unit)
ANSI C
applet
`appletviewer` command
application
arithmetic and logic unit (ALU)
array
assembly language
attribute
Basic
behavior
behavioral design pattern
bytecode
bytecode verifier
C
C standard library
C++
central processing unit (CPU)
class
`.class` file
class libraries
class loader
client/server computing
COBOL
collections
compile phase
compiler
compile-time error
computer
computer program
computer programmer
condition
CPU (central processing unit)
creational design pattern
design pattern
disk
distributed computing
dynamic content
edit phase
editor
encapsulation
event-driven programming
execute phase
execution-time error
fatal runtime error
file server
Fortran
freeware
hardware
high-level language
HotSpot compiler
HTML (Hypertext Markup Language)
IDE (Integrated Development Environment)
information hiding
inheritance
input device
input unit
input/output (I/O)
instance variable
Internet
interpreter
Java
`.java` extension
Java 2 Software Development Kit (J2SDK)
`java` interpreter
Java Virtual Machine
`javac` compiler
JIT (just-in-time) compiler
KIS (keep it simple)
legacy systems
live-code™ approach
load phase
logic error
machine dependent
machine independent
machine language
memory unit
method
Microsoft
Microsoft Internet Explorer Web browser
modeling
multiprocessor
multitasking
multithreading
Netscape Navigator Web browser
nonfatal run-time error
object
object
object-oriented analysis and design (OOAD)
object-oriented design (OOD)
object-oriented programming (OOP)
open source
output device
output unit
Pascal
personal computing
platforms
portability
primary memory
problem statement
procedural programming
programming language
SELF-REVIEW EXERCISES

1.1 Fill in the blanks in each of the following statements:
   a) The company that popularized personal computing was __________.
   b) The computer that made personal computing legitimate in business and industry was the __________.
   c) Computers process data under the control of sets of instructions called __________.
   d) The six key logical units of the computer are the __________, __________, __________, __________, __________ and __________.
   e) The three classes of languages discussed in the chapter are __________, __________ and __________
   f) The programs that translate high-level language programs into machine language are called __________.

1.2 Fill in the blanks in each of the following sentences about the Java environment:
   a) The __________ command from the Java 2 Software Development Kit executes a Java applet.
   b) The __________ command from the Java 2 Software Development Kit executes a Java application
   c) The __________ command from the Java 2 Software Development Kit compiles a Java program.
   d) A(n) __________ file is required to invoke a Java applet.
   e) A Java program file must end with the __________ file extension.
   f) When a Java program is compiled, the file produced by the compiler ends with the __________ file extension.
   g) The file produced by the Java compiler contains __________ that are interpreted to execute a Java applet or application.

1.3 Fill in the blanks in each of the following statements:
   a) The __________ allows computer users to locate and view multimedia-based documents on almost any subject over the Internet.
   b) Java __________ typically are stored on your computer and are designed to execute independent of a World Wide Web browsers.
   c) Lists and tables of values are called __________.
   d) The __________ GUI components are written completely in Java.
   e) __________ allows an applet or application to perform multiple activities in parallel.
   f) __________ provide Java programmers with a standard set of data structures for storing and retrieving data and a standard set of algorithms that allow programmers to manipulate the data.
1.4 Fill in the blanks in each of the following statements (based on Sections 1.15 and 1.16):

a) Over the past decade, the software-engineering industry has made significant progress in the field of ________—proven architectures for constructing flexible and maintainable object-oriented software.

b) Objects have the property of ________.

c) Java programmers concentrate on creating their own user-defined types, called ________.

d) Classes can also have relationships with other classes. These relationships are called ________.

e) The process of analyzing and designing a system from an object-oriented point of view is called ________.

ANSWERS TO SELF-REVIEW EXERCISES

1.1 a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, arithmetic and logic unit, central processing unit, secondary storage unit. e) machine languages, assembly languages, high-level languages. f) compilers.

1.2 a) appletviewer. b) java. c) javac. d) HTML. e) .java. f) .class. g) byte-codes.


1.4 a) design patterns. b) information hiding. c) classes. d) associations. e) object-oriented analysis and design (OOAD).

EXERCISES

1.5 Categorize each of the following items as either hardware or software:

a) CPU
b) Java compiler
c) ALU
d) Java interpreter
e) input unit
f) editor

1.6 Why might you want to write a program in a machine-independent language instead of a machine-dependent language? Why might a machine-dependent language be more appropriate for writing certain types of programs?

1.7 Fill in the blanks in each of the following statements:

a) Which logical unit of the computer receives information from outside the computer for use by the computer? ________.

b) The process of instructing the computer to solve specific problems is called ________.

c) What type of computer language uses English-like abbreviations for machine language instructions? ________.

d) Which logical unit of the computer sends information that has already been processed by the computer to various devices so that the information may be used outside the computer? ________.

e) Which logical unit of the computer retains information? ________.

f) Which logical unit of the computer performs calculations? ________.

g) Which logical unit of the computer makes logical decisions? ________.

h) The level of computer language most convenient to the programmer for writing programs quickly and easily is ________.
i) The only language that a computer can directly understand is called that computer’s ________.

j) Which logical unit of the computer coordinates the activities of all the other logical units? ________.

1.8 Distinguish between the terms fatal error and nonfatal error. Why might you prefer to experience a fatal error rather than a nonfatal error?

1.9 Fill in the blanks in each of the following statements:
   a) Java ________ are designed to be transported over the Internet and executed in World Wide Web browsers.
   b) ________ programming causes a program to perform a task in response to user interactions with graphical user interface (GUI) components.
   c) Java’s graphics capabilities are ________ and, hence portable.
   d) The standard ________ can be used to provide identical user interfaces across all computer platforms.
   e) Languages that cannot perform multiple activities in parallel are called ________ languages or ________ languages.
   f) Aggregations of data such as linked lists, stacks, queues and trees are called ________.

1.10 Fill in the blanks in each of the following statements (based on Sections 1.15 and 1.16):
   a) ________ design patterns describe techniques to instantiate objects (or groups of objects).
   b) The ________ is now the most widely used graphical representation scheme for modeling object-oriented systems.
   c) Java classes contain ________ (which implement class behaviors) and ________ (which implement class data).
   d) ________ design patterns allow designers to organize classes and objects into larger structures.
   e) ________ design patterns assign responsibilities to objects.
   f) In Java, the unit of programming is the ________, from which ________ are eventually instantiated.
Introduction to Java Applications

Objectives

- To be able to write simple Java applications.
- To be able to use input and output statements.
- To become familiar with primitive data types.
- To understand basic memory concepts.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To be able to write decision-making statements.
- To be able to use relational and equality operators.

Comment is free, but facts are sacred.
C. P. Scott

The creditor hath a better memory than the debtor.
James Howell

When faced with a decision, I always ask, “What would be the most fun?”
Peggy Walker

He has left his body to science—and science is contesting the will.
David Frost

Classes struggle, some classes triumph, others are eliminated.
Mao Zedong

Equality, in a social sense, may be divided into that of condition and that of rights.
James Fenimore Cooper
2.1 Introduction

The Java language facilitates a disciplined approach to computer program design. We now introduce Java programming and present examples that illustrate several important features of Java. Each example is analyzed one line at a time. In this chapter and Chapter 3, we present two program types in Java—applications and applets. In Chapter 4 and Chapter 5, we present a detailed treatment of program development and program control in Java.

2.2 A First Program in Java: Printing a Line of Text

Java uses notations that may appear strange to nonprogrammers. We begin by considering a simple application that displays a line of text. An application is a program that executes using the java interpreter (discussed later in this section). The program and its output are shown in Fig. 2.1.

This program illustrates several important features of the Java language. We consider each line of the program in detail. Each program we present in this book has line numbers included for the reader’s convenience; line numbers are not part of actual Java programs. Line 9 does the “real work” of the program, namely displaying the phrase Welcome to Java Programming! on the screen. But let us consider each line in order. Line 1,

// Fig. 2.1: Welcome1.java

begins with //, indicating that the remainder of the line is a comment. Programmers insert comments to document programs and improve program readability. Comments also help other people read and understand a program. Comments do not cause the computer to perform any action when the program is run. The Java compiler ignores comments. We begin every program with a comment indicating the figure number and file name (line 1).
Good Programming Practice 2.1

Use comments to clarify difficult concepts used in a program.

A comment that begins with // is called a single-line comment, because the comment terminates at the end of the current line. A // comment can also begin in the middle of a line and continue until the end of that line.

Multiple-line comments can be written in two other forms. For example,

/* This is a multiple
 line comment. It can be
 split over many lines */

is a comment that can spread over several lines. This type of comment begins with delimiter /* and ends with delimiter */; this type of comment may be called a multiple-line comment. All text between the delimiters of the comment is ignored by the compiler. A similar form of comment called a documentation comment is delimited by /** and */.

Common Programming Error 2.1

Forgetting one of the delimiters of a multiple-line comment is a syntax error.

Java absorbed comments delimited with /* and */ from the C programming language and single-line comments delimited with // from the C++ programming language. Java programmers generally use C++-style single-line comments in preference to C-style comments. Throughout this book, we use C++-style single-line comments. The documentation comment syntax (/** and */) is special to Java. It enables programmers to embed documentation for their programs directly in the programs. The javadoc utility program (provided by Sun Microsystems with the Java 2 Software Development Kit) reads those comments from the program and uses them to prepare your program’s documentation. There are subtle issues to using javadoc-style comments properly. We do not use javadoc-style comments in the programs presented in this book. However, javadoc-style comments are explained thoroughly in Appendix F.
Line 2,

```java
// A first program in Java.
```

is a single-line comment that describes the purpose of the program.

**Good Programming Practice 2.2**

> Every program should begin with a comment describing the purpose of the program.

Line 3 is simply a blank line. Programmers use blank lines and space characters to make programs easier to read. Together, blank lines, space characters and tab characters are known as *white space.* (Space characters and tabs are known specifically as *white-space characters.*) Such characters are ignored by the compiler. We discuss conventions for using white-space characters in this chapter and the next several chapters, as these spacing conventions are needed in may Java programs.

**Good Programming Practice 2.3**

> Use blank lines, space characters and tab characters to enhance program readability.

Line 4,

```java
public class Welcome1 {
```

begins a *class definition* for class *Welcome1.* Every program in Java consists of at least one class definition that is defined by you—the programmer. These classes are known as *programmer-defined classes,* or *user-defined classes.* The *class* keyword introduces a class definition in Java and is immediately followed by the *class name* (*Welcome1* in this program). Keywords (or *reserved words*) are reserved for use by Java (we discuss the various keywords throughout the text) and are always spelled with all lowercase letters. The complete list of Java keywords is shown in Fig. 4.2.

By convention, all class names in Java begin with a capital letter and have a capital letter for every word in the class name (e.g., *SampleClassName*). The name of the class is called an *identifier.* An identifier is a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces. Some valid identifiers are *Welcome1, $value, _value, m_inputField1* and *button7.* The name *7button* is not a valid identifier, because it begins with a digit, and the name *input field* is not a valid identifier, because it contains a space. Java is *case sensitive*—i.e., uppercase and lowercase letters are different, so *a1* and *A1* are different identifiers.

**Common Programming Error 2.2**

> Java is case sensitive. Not using the proper uppercase and lowercase letters for an identifier is normally a syntax error.

**Good Programming Practice 2.4**

> By convention, you should always begin a class name with a capital letter.

**Good Programming Practice 2.5**

> When reading a Java program, look for identifiers that start with capital letters. These identifiers normally represent Java classes.
Software Engineering Observation 2.1

Avoid using identifiers that contain dollar signs ($), as the compiler often uses dollar signs to create identifier names.

In Chapter 2 through Chapter 7, every class we define begins with the public keyword. For now, we will simply require this keyword. The public keyword is discussed in detail in Chapter 8. Also in that chapter, we discuss classes that do not begin with keyword public. [Note: Several times early in this text, we ask you to mimic certain Java features we introduce as you write your own Java programs. We specifically do this when it is not yet important for you to know all of the details of a feature in order for you to use that feature in Java. All programmers initially learn how to program by mimicking what other programmers have done before them. For each detail we ask you to mimic, we indicate where the full discussion will be presented later in the text.]

When you save your public class definition in a file, the file name must be the class name followed by the “.java” file-name extension. For our application, the file name is Welcome1.java. All Java class definitions are stored in files ending with the file-name extension “.java.”

Common Programming Error 2.3

It is an error for a public class if the file name is not identical to the class name (plus the .java extension) in terms of both spelling and capitalization. Therefore, it is also an error for a file to contain two or more public classes.

Common Programming Error 2.4

It is an error not to end a file name with the .java extension for a file containing an application’s class definition. If the extension is missing, the Java compiler will not be able to compile the class definition.

A left brace (at the end of line 4), {, begins the body of every class definition. A corresponding right brace (in line 13 in this program), }, must end each class definition. Notice that lines 6–11 are indented. This indentation is one of the spacing conventions mentioned earlier. We define each spacing convention as a Good Programming Practice.

Good Programming Practice 2.6

Whenever you type an opening left brace, {, in your program, immediately type the closing right brace, }, then reposition the cursor between the braces to begin typing the body. This practice helps prevent errors due to missing braces.

Good Programming Practice 2.7

Indent the entire body of each class definition one “level” of indentation between the left brace, {, and the right brace, }, that define the body of the class. This format emphasizes the structure of the class definition and helps make the class definition easier to read.

Good Programming Practice 2.8

Set a convention for the indent size you prefer, and then uniformly apply that convention. The Tab key may be used to create indents, but tab stops may vary between editors. We recommend using three spaces to form a level of indent.

Common Programming Error 2.5

If braces do not occur in matching pairs, the compiler indicates an error.
Line 5 is a blank line, inserted for program readability. Line 6,

```java
// main method begins execution of Java application
```

is a single-line comment indicating the purpose of lines 6–11 of the program.

Line 7,

```java
public static void main( String args[] )
```

is a part of every Java application. Java applications begin executing at `main`. The parentheses after `main` indicate that `main` is a program building block called a **method**. Java class definitions normally contain one or more methods. For a Java application class, exactly one of those methods must be called `main` and must be defined as shown on line 7; otherwise, the **Java** interpreter will not execute the application. Methods are able to perform tasks and return information when they complete their tasks. The **void** keyword indicates that this method will perform a task (displaying a line of text, in this program), but will not return any information when it completes its task. Later, we will see that many methods return information when they complete their task. Methods are explained in detail in Chapter 6. For now, simply mimic `main`’s first line in your Java applications.

The left brace, `{`, on line 8 begins the **body of the method definition**. A corresponding right brace, `}`, must end the method definition’s body (line 11 of the program). Notice that the line in the body of the method is indented between the braces.

Good Programming Practice 2.9

- **Indent the entire body of each method definition one “level” of indentation between the left brace, {, and the right brace, }, that define the body of the method. This format makes the structure of the method stand out and helps make the method definition easier to read.**

Line 9,

```java
System.out.println( "Welcome to Java Programming!" );
```

instructs the computer to perform an **action**, namely to print the **string** of characters contained between the double quotation marks. A string is sometimes called a **character string**, a **message** or a **string literal**. We refer to characters between double quotation marks generically as **strings**. White-space characters in strings are not ignored by the compiler.

**System.out** is known as the **standard output object**. **System.out** allows Java applications to display strings and other types of information in the **command window** from which the Java application executes. In Microsoft Windows 95/98/ME, the command window is the **MS-DOS prompt**. In Microsoft Windows NT/2000, the command window is the **Command Prompt** (**cmd.exe**). In UNIX, the command window is normally called a **command window**, a **command tool**, a **shell tool** or a **shell**. On computers running an operating system that does not have a command window (such as a Macintosh), the **java** interpreter normally displays a window containing the information the program displays.

Method **System.out.println** displays (or prints) a **line** of text in the command window. When **System.out.println** completes its task, it automatically positions the **output cursor** (the location where the next character will be displayed) to the beginning of the next line in the command window. (This move of the cursor is similar to you pressing the **Enter** key when typing in a text editor—the cursor appears at the beginning of the next line in your file.)
The entire line, including `System.out.println`, its argument in the parentheses (the string) and the semicolon (;), is a statement. Every statement must end with a semicolon (also known as the statement terminator). When the statement on line 9 of our program executes, it displays the message **Welcome to Java Programming!** in the command window.

**Common Programming Error 2.6**

Omitting the semicolon at the end of a statement is a syntax error. A syntax error occurs when the compiler cannot recognize a statement. The compiler normally issues an error message to help the programmer identify and fix the incorrect statement. Syntax errors are violations of the language rules. Syntax errors are also called compile errors, compile-time errors or compilation errors, because the compiler detects them during the compilation phase. You will be unable to execute your program until you correct all of the syntax errors in it.

**Testing and Debugging Tip 2.1**

When the compiler reports a syntax error, the error may not be on the line number indicated by the error message. First, check the line for which the error was reported. If that line does not contain syntax errors, check the preceding several lines in the program.

Some programmers find it difficult when reading and/or writing a program to match the left and right braces ({ and }) that delimit the body of a class definition or a method definition. For this reason, some programmers prefer to include a single-line comment after a closing right brace (}) that ends a method definition and after a closing right brace that ends a class definition. For example, line 11,

```java
} // end method main
```

specifies the closing right brace (}) of method `main`, and line 13,

```java
} // end class Welcome1
```

specifies the closing right brace (}) of class `Welcome1`. Each comment indicates the method or class that the right brace terminates. We use such comments through Chapter 6 to help beginning programmers determine where each program component terminates. After Chapter 6, we use such comments when pairs of braces contain many statements, which makes the closing braces difficult to identify.

**Good Programming Practice 2.10**

Some programmers prefer to follow the closing right brace (}) of a method body or class definition with a single-line comment indicating the method or class definition to which the brace belongs. This comment improves program readability.

### 2.2.1 Compiling and Executing your First Java Application

We are now ready to compile and execute our program. To compile the program, we open a command window, change to the directory where the program is stored and type

```
javac Welcome1.java
```

If the program contains no syntax errors, the preceding command creates a new file called `Welcome1.class` containing the Java bytecodes that represent our application. These bytecodes will be interpreted by the `java` interpreter when we tell it to execute the program, as shown in the Microsoft Windows 2000 **Command Prompt** of Fig. 2.2.
In the command prompt of Figure 2.2, we typed

```
java Welcome1
```

to launch the `java` interpreter and indicate that it should load the “.class” file for class `Welcome1`. Note that the “.class” file-name extension is omitted from the preceding command; otherwise the interpreter will not execute the program. The interpreter automatically calls method `main`. Next, the statement on line 7 of `main` displays “Welcome to Java Programming!”

**Testing and Debugging Tip 2.2**

The Java compiler generates syntax error messages when the syntax of a program is incorrect. When you are learning how to program, sometimes it is helpful to “break” a working program so you can see the error messages produced by the compiler. Then, when you encounter that error message again, you will have an idea of the error’s cause. Try removing a semicolon or curly brace from the program of Fig. 2.1, then recompile the program to see the error messages generated by the omission.

### 2.3 Modifying Our First Java Program

This section continues our introduction to Java programming with two examples that modify the example in Fig. 2.1 to print text on one line by using multiple statements and to print text on several lines by using a single statement.

#### 2.3.1 Displaying a Single Line of Text with Multiple Statements

`Welcome to Java Programming!` can be displayed using several methods. Class `Welcome2`, shown in Fig. 2.3, uses two statements to produce the same output as that shown in Fig. 2.1.

Most of the program is identical to that of Fig. 2.1, so we discuss only the changes here. Line 2,

```
// Printing a line of text with multiple statements.
```

is a single-line comment stating the purpose of this program. Line 4 begins the definition of class `Welcome2`.

```
System.out.print( "Welcome to " );
System.out.println( "Java Programming!" );
```
display one line of text in the command window. The first statement uses System.out's method print to display a string. The difference between print and println is that, after displaying its argument, print does not position the output cursor at the beginning of the next line in the command window; the next character the program displays in the command window will appear immediately after the last character that print displays. Thus, line 10 positions the first character in its argument, "J," immediately after the last character that line 9 displays (the space character at the end of the string on line 9). Each print or println statement resumes displaying characters from where the last print or println statement stopped displaying characters.

### 2.3.2 Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using newline characters. Newline characters are “special characters” that indicate to System.out’s print and println methods when they should position the output cursor to the beginning of the next line in the command window. Figure 2.4 outputs four lines of text, using newline characters to determine when to begin each new line.

Most of the program is identical to those of Fig. 2.1 and Fig. 2.3, so we discuss only the changes here. Line 2,

// Printing multiple lines of text with a single statement.

is a single-line comment stating the purpose of this program. Line 4 begins the definition of class Welcome3.

Line 9,

System.out.println( "Welcome\nto\nJava\nProgramming!" );

displays four separate lines of text in the command window. Normally, the characters in a string are displayed exactly as they appear in the double quotes. Notice, however, that the
two characters \ and n are not printed on the screen. The backslash (\) is called an escape character. It indicates that a “special character” is to be output. When a backslash appears in a string of characters, Java combines the next character with the backslash to form an escape sequence. The escape sequence \n is the newline character. When a newline character appears in a string being output with System.out, the newline character causes the screen’s output cursor to move to the beginning of the next line in the command window. Some other common escape sequences are listed in Fig. 2.5.

```
// Fig. 2.4: Welcome3.java
// Printing multiple lines of text with a single statement.
public class Welcome3 {
  // main method begins execution of Java application
  public static void main( String args[] )
  {
    System.out.println( "Welcome\nto\nJava\nProgramming!" );
  }  // end method main
} // end class Welcome3
```

Welcome
to
Java
Programming!

**Fig. 2.4** Printing multiple lines of text with a single statement.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline. Position the screen cursor to the beginning of the next line.</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab. Move the screen cursor to the next tab stop.</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.</td>
</tr>
<tr>
<td>\</td>
<td>Backslash. Used to print a backslash character.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote. Used to print a double-quote character. For example,</td>
</tr>
<tr>
<td></td>
<td>System.out.println( &quot;&quot;in quotes&quot;&quot; );</td>
</tr>
<tr>
<td></td>
<td>displays</td>
</tr>
<tr>
<td></td>
<td>&quot;in quotes&quot;</td>
</tr>
</tbody>
</table>

**Fig. 2.5** Some common escape sequences.
2.4 Displaying Text in a Dialog Box

Although the first several programs presented in this chapter display output in the command window, many Java applications use windows or dialog boxes (also called dialogs) to display output. For example, World Wide Web browsers such as Netscape Navigator or Microsoft Internet Explorer display Web pages in their own windows. Email programs typically allow you to type and read messages in a window provided by the email program. Typically, dialog boxes are windows in which programs display important messages to the user of the program. Java’s class `JOptionPane` provides prepackaged dialog boxes that enable programs to display messages to users. Figure 2.6 displays the same string as in Fig. 2.4 in a predefined dialog box known as a message dialog.

One of the great strengths of Java is its rich set of predefined classes that programmers can reuse rather than “reinventing the wheel.” We use many of these classes throughout the book. Java’s numerous predefined classes are grouped into categories of related classes called packages. The packages are referred to collectively as the Java class library, or the Java applications programming interface (Java API). The packages of the Java API are split into core packages and extension packages. The names of the packages begin with either “java” (core packages) or “javax” (extension packages). Many of the core and extension packages are included as part of the Java 2 Software Development Kit. We overview these included packages in Chapter 6. As Java continues to evolve, new packages are developed as extension packages. These extensions often can be downloaded from `java.sun.com` and used to enhance Java’s capabilities. In this example, we use class `JOptionPane`, which Java defines for us in package `javax.swing`.

Line 4,

```
// Java extension packages
```

is a single-line comment indicating the section of the program in which we specify `import` statements for classes in Java’s extension packages. In every program that specifies `import` statements, we separate the import statements into the following groups: Java core packages (for package names starting with `java`), Java extension packages (for package names starting with `javax`) and Deitel packages (for our own packages defined later in the book).

Line 5,

```
import javax.swing.JOptionPane;  // import class JOptionPane
```

is an `import` statement. The compiler uses `import` statements to identify and load classes used in a Java program. When you use classes from the Java API, the compiler attempts to ensure that you use them correctly. The `import` statements help the compiler locate the classes you intend to use. For each new class we use from the Java API, we indicate the package in which you can find that class. This package information is important. It helps you locate descriptions of each package and class in the Java API documentation. A Web-based version of this documentation can be found at

```
java.sun.com/j2se/1.3/docs/api/index.html
```

Also, you can download this documentation to your own computer from

```
java.sun.com/j2se/1.3/docs.html
```
We will provide an overview of the use of this documentation with the downloads and re-

**Common Programming Error 2.7**

All `import` statements must appear before the class definition. Placing an `import` state-
ment inside a class definition’s body or after a class definition is a syntax error.

Line 5 tells the compiler to load the `JOptionPane` class from the `javax.swing` package. This package contains many classes that help Java programmers define *graphical user interfaces* (GUIs) for their application. *GUI components* facilitate data entry by the user of your program and formatting or presentation of data outputs to the user of your pro-
gram. For example, Fig. 2.7 contains a Netscape Navigator window. In the window, there is a bar containing *menus* (*File, Edit, View*, etc.), called a *menu bar*. Below the menu bar is a set of *buttons* that each have a defined task in Netscape Navigator. Below the buttons there is a *text field* in which the user can type the name of the World Wide Web site to visit. The menus, buttons and text fields are part of Netscape Navigator’s GUI. They enable you to interact with the Navigator program. Java contains classes that implement the GUI com-
ponents described here and others that will be described in Chapter 12, Basic Graphical User Interface Components, and Chapter 13, Advanced Graphical User Interface Compo-

```java
// Fig. 2.6: Welcome4.java
// Printing multiple lines in a dialog box

// Java extension packages
import javax.swing.JOptionPane;  // import class JOptionPane

public class Welcome4 {
    // main method begins execution of Java application
    public static void main( String args[] )
    {
        JOptionPane.showMessageDialog(
            null, "Welcome\nto\nJava\nProgramming!" );

        System.exit( 0 );  // terminate application
    }
}
```

Fig. 2.6 Displaying multiple lines in a dialog box.
In method `main` of Fig. 2.6, lines 12–13,

```
JOptionPane.showMessageDialog(
    null, "Welcome
to
Java
Programming!" );
```

indicate a call to method `showMessageDialog` of class `JOptionPane`. The method requires two arguments. When a method requires multiple arguments, the arguments are separated with commas (,). Until we discuss `JOptionPane` in detail in Chapter 13, the first argument will always be the keyword `null`. The second argument is the string to display. The first argument helps the Java application determine where to position the dialog box. When the first argument is `null`, the dialog box appears in the center of the computer screen. Most applications you use on your computer execute in their own window (e.g., email programs, Web browsers and word processors). When such an application displays a dialog box, it normally appears in the center of the application window, which is not necessarily the center of the screen. Later in this book, you will see more elaborate applications in which the first argument to method `showMessageDialog` will cause the dialog box to appear in the center of the application window, rather than the center of the screen.

**Good Programming Practice 2.11**

Place a space after each comma (,) in an argument list, to make programs more readable.

Method `JOptionPane.showMessageDialog` is a special method of class `JOptionPane` called a static method. Such methods are always called by using their class name followed by a dot operator (.) and the method name, as in

```
ClassName . methodName( arguments )
```

Many of the predefined methods we introduce early in this book are static methods. We ask you to mimic this syntax for calling static methods until we discuss them in detail in Chapter 8, Object-Based Programming.
Executing the statement at lines 12–13 displays the dialog box in Fig. 2.8. The title bar of the dialog contains the string Message, to indicate that the dialog is presenting a message to the user. The dialog box automatically includes an OK button that allows the user to dismiss (hide) the dialog by pressing the button. This is accomplished by positioning the mouse cursor (also called the mouse pointer) over the OK button and clicking the left mouse button.

Remember that all statements in Java end with a semicolon (;). Therefore, lines 12–13 represent one statement. Java allows large statements to be split over many lines. However, you cannot split a statement in the middle of an identifier or in the middle of a string.

**Common Programming Error 2.8**

Splitting a statement in the middle of an identifier or a string is a syntax error.

Line 15,

```java
System.exit( 0 );  // terminate application
```

uses static method `exit` of class `System` to terminate the application. In any application that displays a graphical user interface, this line is required in order to terminate the application. Notice once again the syntax used to call the method—the class name (`System`), a dot (.) and the method name (`exit`). Remember that identifiers starting with capital letters normally represent class names. So, you can assume that `System` is a class. Class `System` is part of the package `java.lang`. Notice that class `System` is not imported with an import statement at the beginning of the program. By default, package `java.lang` is imported in every Java program. Package `java.lang` is the only package in the Java API for which you are not required to provide an import statement.

The argument `0` to method `exit` indicates that the application has terminated successfully. (A nonzero value normally indicates that an error has occurred.) This value is passed to the command window that executed the program. The argument is useful if the program is executed from a batch file (on Windows 95/98/ME/NT/2000 systems) or a shell script (on UNIX/Linux systems). Batch files and shell scripts often execute several programs in sequence. When the first program ends, the next program begins execution automatically. It is possible to use the argument to method `exit` in a batch file or shell script to determine whether other programs should execute. For more information on batch files or shell scripts, see your operating system’s documentation.
Common Programming Error 2.9

Forgetting to call `System.exit` in an application that displays a graphical user interface prevents the program from terminating properly. This omission normally results in the command window preventing you from typing any other commands. Chapter 14 discusses in more detail the reason that `System.exit` is required in GUI-based applications.

2.5 Another Java Application: Adding Integers

Our next application inputs two integers (whole numbers, like –22, 7 and 1024) typed by a user at the keyboard, computes the sum of the values and displays the result. This program uses another predefined dialog box from class `JOptionPane` called an input dialog that allows the user to input a value for use in the program. The program also uses a message dialog to display the sum of the integers. Figure 2.9 shows the application and sample screen captures.

```java
// Fig. 2.9: Addition.java
// An addition program.

// Java extension packages
import javax.swing.JOptionPane; // import class JOptionPane

public class Addition {

    // main method begins execution of Java application
    public static void main( String args[] )
    {
        String firstNumber;  // first string entered by user
        String secondNumber; // second string entered by user
        int number1;          // first number to add
        int number2;          // second number to add
        int sum;              // sum of number1 and number2

        // read in first number from user as a String
        firstNumber = JOptionPane.showInputDialog( "Enter first integer" );

        // read in second number from user as a String
        secondNumber = JOptionPane.showInputDialog( "Enter second integer" );

        // convert numbers from type String to type int
        number1 = Integer.parseInt( firstNumber );
        number2 = Integer.parseInt( secondNumber );

        // add the numbers
        sum = number1 + number2;

        // display the results
        JOptionPane.showMessageDialog( null, "The sum is " + sum, "Results", JOptionPane.PLAIN_MESSAGE );
    }
}
```

Fig. 2.9 An addition program “in action” (part 1 of 2).
Lines 1 and 2,

// Fig. 2.9: Addition.java
// An addition program.

are single-line comments stating the figure number, file name and purpose of the program.

Line 4,

// Java extension packages

is a single-line comment specifying that the next line imports a class from the Java extension packages.

Line 5,

import javax.swing.JOptionPane;  // import class JOptionPane

indicates that the compiler should load class JOptionPane for use in this application.

As stated earlier, every Java program consists of at least one class definition. Line 7,

public class Addition {

begins the definition of class Addition. The file name for this public class must be Addition.java.

Remember that all class definitions start with an opening left brace (at the end of line 7), {, and end with a closing right brace, } (in line 42).

As stated earlier, every application begins execution with method main (lines 10–40). The left brace (line 11) marks the beginning of main’s body and the corresponding right brace (line 36) marks the end of main’s body.

Lines 12–13,

String firstNumber;   // first string entered by user
String secondNumber;  // second string entered by user
are declarations. The words `firstNumber` and `secondNumber` are the names of variables. A variable is a location in the computer’s memory where a value can be stored for use by a program. All variables must be declared with a name and a data type before they can be used in a program. This declaration specifies that the variables `firstNumber` and `secondNumber` are data of type `String` (located in package `java.lang`), which means that the variables will hold strings. A variable name can be any valid identifier. Like statements, declarations end with a semicolon (`;`). Notice the single-line comments at the end of each line. This use and placement of the comments is a common practice used by programmers to indicate the purpose of each variable in the program.

**Good Programming Practice 2.12**

Choosing meaningful variable names helps a program to be self-documenting (i.e., it becomes easier to understand the program simply by reading it rather than by reading manuals or viewing an excessive number of comments).

**Good Programming Practice 2.13**

By convention, variable-name identifiers begin with a lowercase letter. As with class names, every word in the name after the first word should begin with a capital letter. For example, identifier `firstNumber` has a capital `N` in its second word, `Number`.

**Good Programming Practice 2.14**

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.

**Software Engineering Observation 2.2**

Java automatically imports classes from package `java.lang`, such as class `String`. Therefore, `import` statements are not required for classes in package `java.lang`.

Declarations can be split over several lines, with each variable in the declaration separated by a comma (i.e., a comma-separated list of variable names). Several variables of the same type may be declared in one declaration or in multiple declarations. Lines 12–13 can also be written as follows:

```java
String firstNumber,   // first string entered by user
    secondNumber;  // second string entered by user
```

Lines 14–16,

```java
int number1;       // first number to add
int number2;       // second number to add
int sum;           // sum of number1 and number2
```

declare that variables `number1`, `number2` and `sum` are data of type `int`, which means that these variables will hold integer values (whole numbers such as 7, –11, 0 and 31,914). We will soon discuss the data types `float` and `double`, for specifying real numbers (numbers with decimal points, such as 3.4, 0.0 and –11.19), and variables of type `char`, for specifying character data. A `char` variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single special character (such as `x`, `$`, `7` and `*`) and escape sequences (such as the newline character, `\n`). Java is capable of representing characters from many other spoken languages. Types such as `int`, `double` and `char` are often called primitive data types, or built-in data types. Primitive-type names are keywords;
thus, they must appear in all lowercase letters. Chapter 4 summarizes the eight primitive
types (boolean, char, byte, short, int, long, float and double).

Line 18 is a single-line comment indicating that the next statement reads the first
number from the user. Lines 19–20,

```java
firstNumber =
    JOptionPane.showInputDialog( "Enter first integer" );
```

reads from the user a String representing the first of the two integers to add. Method
JOptionPane.showInputDialog displays the input dialog in Fig. 2.10.

The argument to showInputDialog indicates what the user should type in the text
field. This message is called a prompt, because it directs the user to take a specific action.
The user types characters in the text field, and then clicks the OK button or presses the
Enter key to return the string to the program. (If you type and nothing appears in the text
field, position the mouse pointer in the text field and click the left mouse button to activate
the text field.) Unfortunately, Java does not provide a simple form of input that is analogous
to displaying output in the command window with System.out’s method print and
println. For this reason, we normally receive input from a user through a GUI compo-
nent (an input dialog box in this program).

Technically, the user can type anything in the text field of the input. Our program
assumes that the user follows directions and enters a valid integer value. In this program, if
the user either types a noninteger value or clicks the Cancel button in the input dialog, a
runtime logic error will occur. Chapter 14, Exception Handling, discusses how to make
your programs more robust by enabling them to handle such errors. This is also known as
making your program fault tolerant.

The result of JOptionPane method showInputDialog (a String containing
the characters typed by the user) is given to variable firstNumber by using the assign-
ment operator, =. The statement (lines 19–20) is read as “firstNumber gets the value
of JOptionPane.showInputDialog( "Enter first integer" ).” The =
operator is called a binary operator, because it has two operands: firstNumber and the
result of the expression JOptionPane.showInputDialog( "Enter first
integer" ). This whole statement is called an assignment statement, because it assigns
a value to a variable. The expression to the right side of the assignment operator, =, is
always evaluated first. In this case, the program calls method showInputDialog, and
the value input by the user is assigned to firstNumber.

---

**Fig. 2.10** Input dialog box.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
Line 22 is a single-line comment indicating that the next statement reads the second number from the user.

Lines 23–24,

```java
secondNumber =
    JOptionPane.showInputDialog( "Enter second integer" );
```

display an input dialog in which the user types a `String` representing the second of the two integers to add.

Lines 27–28,

```java
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```

convert the two `String`s input by the user to `int` values that the program can use in a calculation. Method `Integer.parseInt` (a `static` method of class `Integer`) converts its `String` argument to an integer. Class `Integer` is defined in package `java.lang`. Line 27 assigns the `int` (integer) value that `Integer.parseInt` returns to variable `number1`. Line 28 assigns the `int` (integer) value that `Integer.parseInt` returns to variable `number2`.

Line 31,

```java
sum = number1 + number2;
```

is an assignment statement that calculates the sum of the variables `number1` and `number2` and assigns the result to variable `sum` by using the assignment operator, `=`. The statement is read as, “`sum` gets the value of `number1 + number2`.” Most calculations are performed in assignment statements. When the program encounters the addition operation, it uses the values stored in the variables `number1` and `number2` to perform the calculation. In the preceding statement, the addition operator is a binary operator: its two operands are `number1` and `number2`.

### Good Programming Practice 2.15

*Place spaces on either side of a binary operator. This format makes the operator stand out and makes the program more readable.*

After the calculation has been performed, lines 34–36,

```java
JOptionPane.showMessageDialog(
    null, "The sum is " + sum, "Results",
    JOptionPane.PLAIN_MESSAGE );
```

use method `JOptionPane.showMessageDialog` to display the result of the addition. This new version of `JOptionPane` method `showMessageDialog` requires four arguments. As in Fig. 2.6, the `null` first argument indicates that the message dialog will appear in the center of the screen. The second argument is the message to display. In this case, the second argument is the expression

"The sum is " + sum
which uses the operator + to “add” a String (the literal "The sum is ") and the value of variable sum (the int variable containing the result of the addition on line 31). Java has a version of the + operator for string concatenation that concatenates a String and a value of another data type (including another String); the result of this operation is a new (and normally longer) String. If we assume that sum contains the integer value 117, the expression evaluates as follows:

1. Java determines that the two operands of the + operator (the string "The sum is ") and the integer sum) are of different types and one of them is a String.
2. Java converts sum to a String.
3. Java appends the String representation of sum to the end of "The sum is ", resulting in the String "The sum is 117".

Method showMessageDialog displays the resulting String in the dialog box. Note that the automatic conversion of integer sum occurs only because the addition operation concatenates the String literal "The sum is " and sum. Also, note that the space between is and 117 is part of the string "The sum is ". String concatenation is discussed in detail in Chapter 10, “Strings and Characters.”

**Common Programming Error 2.10**

Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. For example, assuming that integer variable y has the value 5, the expression "y+2=" + y + 2 results in the string "y+2=5", not "y+2=7", because first the value of y is concatenated with the string "y+2=" and then the value 2 is concatenated with the new larger string "y+2=5". The expression "y+2=" +(y+2) produces the desired result.

The third and fourth arguments of method showMessageDialog in Fig. 2.9 represent the string that should appear in the dialog box’s title bar and the dialog box type, respectively. The fourth argument—JOptionPane.PLAIN_MESSAGE—is a value indicating the type of message dialog to display. This type of message dialog does not display an icon to the left of the message. Figure 2.11 illustrates the second and third arguments and shows that there is no icon in the window.

The message dialog types are shown in Fig. 2.12. All message dialog types except PLAIN_MESSAGE dialogs display an icon to the user indicating the type of message.
2.6 Memory Concepts

Variable names such as `number1`, `number2` and `sum` actually correspond to locations in the computer's memory. Every variable has a name, a type, a size and a value.

In the addition program of Fig. 2.9, when the statement

```java
number1 = Integer.parseInt( firstNumber );
```

executes, the string previously typed by the user in the input dialog and stored in `firstNumber` is converted to an `int` and placed into a memory location to which the name `number1` has been assigned by the compiler. Suppose that the user enters the string `45` as the value for `firstNumber`. The program converts `firstNumber` to an `int`, and the computer places that integer value, `45`, into location `number1`, as shown in Fig. 2.13.

Whenever a value is placed in a memory location, the value replaces the previous value in that location. The previous value is destroyed (i.e., lost).

When the statement

```java
number2 = Integer.parseInt( secondNumber );
```

executes, suppose that the user enters the string `72` as the value for `secondNumber`. The program converts `secondNumber` to an `int`, and the computer places that integer value, `72`, into location `number2`. The memory appears as shown in Fig. 2.14.
After the program of Fig. 2.9 obtains values for `number1` and `number2`, it adds the values and places the sum into variable `sum`. The statement

```
sum = number1 + number2;
```

performs the addition and also replaces `sum`’s previous value. After `sum` has been calculated, memory appears as shown in Fig. 2.15. Note that the values of `number1` and `number2` appear exactly as they did before they were used in the calculation of `sum`. These values were used, but not destroyed, as the computer performed the calculation. Thus, when a value is read from a memory location, the process is nondestructive.

### 2.7 Arithmetic

Most programs perform arithmetic calculations. The **arithmetic operators** are summarized in Fig. 2.16. Note the use of various special symbols not used in algebra. The *asterisk* (\( \ast \)) indicates multiplication, and the *percent sign* (\( \% \)) is the *modulus operator*, which is discussed shortly. The arithmetic operators in Fig. 2.16 are binary operators, because they each operate on two operands. For example, the expression `sum + value` contains the binary operator `+` and the two operands `sum` and `value`.

**Integer division** yields an integer quotient; for example, the expression `7 / 4` evaluates to `1`, and the expression `17 / 5` evaluates to `3`. Note that any fractional part in integer division is simply discarded (i.e., truncated)—no rounding occurs. Java provides the modulus operator, `\( \% \)`, that yields the remainder after integer division. The expression `x \% y` yields the remainder after `x` is divided by `y`. Thus, `7 \% 4` yields `3`, and `17 \% 5` yields `2`. This operator is most commonly used with integer operands, but also can be used with other arithmetic types. In later chapters, we consider many interesting applications of the modulus operator, such as determining if one number is a multiple of another. There is no arithmetic operator for exponentiation in Java. Chapter 5 shows how to perform exponentiation in Java. [Note: The modulus operator can be used with both integer and floating-point numbers.]

**Fig. 2.14** Memory locations after storing values for `number1` and `number2`.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>number1</code></td>
<td>45</td>
</tr>
<tr>
<td><code>number2</code></td>
<td>72</td>
</tr>
</tbody>
</table>

**Fig. 2.15** Memory locations after calculating the sum of `number1` and `number2`.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>number1</code></td>
<td>45</td>
</tr>
<tr>
<td><code>number2</code></td>
<td>72</td>
</tr>
<tr>
<td><code>sum</code></td>
<td>117</td>
</tr>
</tbody>
</table>
Arithmetic expressions in Java must be written in \textit{straight-line form} to facilitate entering programs into the computer. Thus, expressions such as “\(a\) divided by \(b\)” must be written as \(a/b\), so that all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to compilers:

\[
\frac{a}{b}
\]

Parentheses are used in Java expressions in the same manner as in algebraic expressions. For example, to multiply \(a\) times the quantity \(b + c\), we write

\[
a \times (b + c)
\]

Java applies the operators in arithmetic expressions in a precise sequence determined by the following \textit{rules of operator precedence}, which are generally the same as those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Thus, \textit{parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer}. Parentheses are at the highest level of precedence. In cases of \textit{nested} or \textit{embedded} parentheses, the operators in the innermost pair of parentheses are applied first.

2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division or modulus operations, the operators are applied from left to right. Multiplication, division and modulus operators have the same level of precedence.

3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

The rules of operator precedence enable Java to apply operators in the correct order. When we say that operators are applied from left to right, we are referring to the \textit{associativity} of the operators. We will see that some operators associate from right to left. Figure 2.17 summarizes these rules of operator precedence. This table will be expanded as additional Java operators are introduced. A complete precedence chart is included in Appendix C.
Now, let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its Java equivalent.

The following is an example of an arithmetic mean (average) of five terms:

Algebra: \[ m = \frac{a + b + c + d + e}{5} \]

Java: \[ m = ( a + b + c + d + e ) / 5; \]

The parentheses are required, because division has higher precedence than that of addition. The entire quantity \((a + b + c + d + e)\) is to be divided by 5. If the parentheses are erroneously omitted, we obtain \(a + b + c + d + e / 5\), which evaluates as \(a + b + c + d + e / 5\), which evaluates as

\[ a + b + c + d + \frac{e}{5} \]

The following is an example of the equation of a straight line:

Algebra: \[ y = mx + b \]

Java: \[ y = m * x + b; \]

No parentheses are required. The multiplication operator is applied first, because multiplication has a higher precedence than that of addition. The assignment occurs last, because it has a lower precedence than that of multiplication and division.

The following example contains modulus (%), multiplication, division, addition and subtraction operations:

Algebra: \[ z = pr\%q + w/x – y \]

Java: \[ z = p * r \% q + w / x - y; \]
The circled numbers under the statement indicate the order in which Java applies the operators. The multiplication, modulus and division operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than that of addition and subtraction. The addition and subtraction operations are evaluated next. These operations are also applied from left to right.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

\[ a \times ( b + c ) + c \times ( d + e ) \]

does not contain nested parentheses. Rather, these parentheses are on the same level.

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial \(( y = ax^2 + bx + c )\):

\[ y = a \times x \times x + b \times x + c; \]

The circled numbers under the preceding statement indicate the order in which Java applies the operators. There is no arithmetic operator for exponentiation in Java; \(x^2\) is represented as \(x \times x\).

Suppose that \(a, b, c\) and \(x\) are initialized as follows: \(a = 2, b = 3, c = 7\) and \(x = 5\). Figure 2.18 illustrates the order in which the operators are applied in the preceding second-degree polynomial.

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. Such unnecessary parentheses are also called redundant parentheses. For example, the preceding assignment statement might be parenthesized as follows:

\[ y = ( a \times x \times x ) + ( b \times x ) + c; \]

**Good Programming Practice 2.16**

*Using parentheses for complex arithmetic expressions, even when the parentheses are not necessary, can make the arithmetic expressions easier to read.*

### 2.8 Decision Making: Equality and Relational Operators

This section introduces a simple version of Java’s *if* structure that allows a program to make a decision based on the truth or falsity of some condition. If the condition is met (i.e., the condition is *true*), the statement in the body of the *if* structure is executed. If the condition is not met (i.e., the condition is *false*), the body statement does not execute. We will see an example shortly.

Conditions in *if* structures can be formed by using the equality operators and relational operators summarized in Fig. 2.19. The relational operators all have the same level of precedence and associate from left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right.
### Fig. 2.18
Order in which a second-degree polynomial is evaluated.

<table>
<thead>
<tr>
<th>Standard algebraic equality or relational operator</th>
<th>Java equality or relational operator</th>
<th>Example of Java condition</th>
<th>Meaning of Java condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$=$</td>
<td>$==$</td>
<td>$x == y$</td>
<td>$x$ is equal to $y$</td>
</tr>
<tr>
<td>$\neq$</td>
<td>$!=$</td>
<td>$x != y$</td>
<td>$x$ is not equal to $y$</td>
</tr>
<tr>
<td>Relational operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$x &gt; y$</td>
<td>$x$ is greater than $y$</td>
</tr>
<tr>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$x &lt; y$</td>
<td>$x$ is less than $y$</td>
</tr>
<tr>
<td>$\geq$</td>
<td>$\geq$</td>
<td>$x \geq y$</td>
<td>$x$ is greater than or equal to $y$</td>
</tr>
<tr>
<td>$\leq$</td>
<td>$\leq$</td>
<td>$x \leq y$</td>
<td>$x$ is less than or equal to $y$</td>
</tr>
</tbody>
</table>

### Fig. 2.19
Equality and relational operators.
Common Programming Error 2.11

It is a syntax error if the operators ==, !=, >= and <= contain spaces between their symbols, as in = =, ! =, > = and < =, respectively.

Common Programming Error 2.12

Reversing the operators !=, >= and <=, as in =!, => and =<, is a syntax error.

Common Programming Error 2.13

Confusing the equality operator, ==, with the assignment operator, =, can be a logic error or a syntax error. The equality operator should be read as “is equal to,” and the assignment operator should be read as “gets” or “gets the value of.” Some people prefer to read the equality operator as “double equals” or “equals equals.”

The next example uses six if structures to compare two numbers input into text fields by the user. If the condition in any of these if statements is true, the assignment statement associated with that if structure executes. The user inputs two values through input dialogs. Next, the program converts the input values to integers and stores them in variables number1 and number2. Then, the program compares the numbers and displays the results of the comparisons in an information dialog. The program and sample outputs are shown in Fig. 2.20.

```java
// Fig. 2.20: Comparison.java
// Compare integers using if structures, relational operators
// and equality operators.

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

public class Comparison {

  // main method begins execution of Java application
  public static void main( String args[] )
  {
    String firstNumber;   // first string entered by user
    String secondNumber;  // second string entered by user
    String result;        // a string containing the output
    int number1;          // first number to compare
    int number2;          // second number to compare

    // read first number from user as a string
    firstNumber = JOptionPane.showInputDialog( "Enter first integer:" );

    // read second number from user as a string
    secondNumber = JOptionPane.showInputDialog( "Enter second integer:" );

    // compare numbers
    if (firstNumber.equals(secondNumber))
      result = "Numbers are equal.
    else
      result = "Numbers are not equal.

    JOptionPane.showMessageDialog( null, result );
  }
}
```

Fig. 2.20 Using equality and relational operators (part 1 of 3).
```java
// convert numbers from type String to type int
number1 = Integer.parseInt(firstNumber);
number2 = Integer.parseInt(secondNumber);

// initialize result to empty String
result = "";

if (number1 == number2)
    result = number1 + " == " + number2;

if (number1 != number2)
    result = number1 + " != " + number2;

if (number1 < number2)
    result = result + "\n" + number1 + " < " + number2;

if (number1 > number2)
    result = result + "\n" + number1 + " > " + number2;

if (number1 <= number2)
    result = result + "\n" + number1 + " <= " + number2;

if (number1 >= number2)
    result = result + "\n" + number1 + " >= " + number2;

// Display results
JOptionPane.showMessageDialog(null, result, "Comparison Results", JOptionPane.INFORMATION_MESSAGE);
System.exit(0); // terminate application
```

Fig. 2.20 Using equality and relational operators (part 2 of 3).
The definition of application class **Comparison** begins at line 8.

```java
public class Comparison {

As discussed previously, method **main** (lines 11–59) begins the execution of every Java application.

Lines 13–17,

```java
String firstNumber;   // first string entered by user
String secondNumber;  // second string entered by user
String result;        // a string containing the output
int number1;          // first number to compare
int number2;          // second number to compare

```java
declare the variables used in method **main**. Note that there are three variables of type **String** and two variables of type **int**. Remember that variables of the same type may be declared in one declaration or in multiple declarations. If more than one name is declared in a declaration, the names are separated by commas (,), as in

```java
String firstNumber, secondNumber, result;
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
or as in

```java
String firstNumber,
    secondNumber,
    result;
```

This is a set of names known as a **comma-separated list**. Once again, notice the comment at the end of each declaration in lines 13–17, indicating the purpose of each variable in the program.

Lines 20–21,

```java
firstNumber =
    JOptionPane.showInputDialog( "Enter first integer:" );
```

use `JOptionPane.showInputDialog` to allow the user to input the first integer value as a string and store it in `firstNumber`.

Lines 24–25,

```java
secondNumber =
    JOptionPane.showInputDialog( "Enter second integer:" );
```

use `JOptionPane.showInputDialog` to allow the user to input the second integer value as a string and store it in `secondNumber`.

Lines 28–29,

```java
number1 = Integer.parseInt( firstNumber );
number2 = Integer.parseInt( secondNumber );
```

convert each string input by the user in the input dialogs to type `int` and assign the values to int variables `number1` and `number2`.

Line 32,

```java
result = "";
```

assigns to `result` the **empty string**—a string containing no characters. Every variable declared in a method (such as `main`) must be **initialized** (given a value) before it can be used in an expression. Because we do not yet know what the final `result` string will be, we assign to `result` the empty string as a temporary initial value.

---

**Common Programming Error 2.14**

Not initializing a variable defined in a method before that variable is used in the method’s body is a syntax error.

Lines 34–35,

```java
if ( number1 == number2 )
    result = result + number1 + " == " + number2;
```

define an **if structure** that compares the values of the variables `number1` and `number2` to determine if they are equal. The **if** structure always begins with keyword **if**, followed by a condition in parentheses. The **if** structure expects one statement in its body. The indentation shown here is not required, but it improves the readability of the program by emphasizing that the statement in line 35 is part of the **if** structure that begins on line 34.
Good Programming Practice 2.17

Indent the statement in the body of an if structure to make the body of the structure stand out and to enhance program readability.

Good Programming Practice 2.18

Place only one statement per line in a program. This format enhances program readability.

In the preceding if structure, if the values of variables number1 and number2 are equal, line 35 assigns to result the value of result + number1 + " == " + number2. As discussed in Fig. 2.9, the + operator in this expression performs string concatenation. For this discussion, we assume that each of the variables number1 and number2 has the value 123. First, the expression converts number1’s value to a string and appends it to result (which currently contains the empty string) to produce the string "123". Next, the expression appends " == " to "123" to produce the string "123 == ". Finally, the expression appends number2 to "123 == " to produce the string "123 == 123". The String result becomes longer as the program proceeds through the if structures and performs more concatenations. For example, given the value 123 for both number1 and number2 in this discussion, the if conditions at lines 46–47 (<=) and 49–50 (>=) are also true. So, the program displays the result

123 == 123
123 <= 123
123 >= 123

in a message dialog.

Common Programming Error 2.15

Replacing operator == in the condition of an if structure, such as if (x == 1), with operator =, as in if (x = 1), is a syntax error.

Common Programming Error 2.16

Forgetting the left and right parentheses for the condition in an if structure is a syntax error. The parentheses are required.

Notice that there is no semicolon (;) at the end of the first line of each if structure. Such a semicolon would result in a logic error at execution time. For example,

```java
if ( number1 == number2 ); // logic error
result = result + number1 + " == " + number2;
```

would actually be interpreted by Java as

```java
if ( number1 == number2 )
{
result = result + number1 + " == " + number2;
```

where the semicolon on the line by itself—called the empty statement—is the statement to execute if the condition in the if structure is true. When the empty statement executes, no task is performed in the program. The program then continues with the assignment statement, which executes regardless of whether the condition is true or false.
Common Programming Error 2.17

Placing a semicolon immediately after the right parenthesis of the condition in an `if` structure is normally a logic error. The semicolon will cause the body of the `if` structure to be empty, so the `if` structure itself will perform no action, regardless of whether its condition is true. Worse yet, the intended body statement of the `if` structure will now become a statement in sequence with the `if` structure and will always be executed.

Notice the use of spacing in Fig. 2.20. Remember that white-space characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to the programmer’s preferences without affecting the meaning of a program. It is incorrect to split identifiers and string literals. Ideally, statements should be kept small, but it is not always possible to do so.

Good Programming Practice 2.19

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.

The chart in Fig. 2.21 shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. Notice that all of these operators, with the exception of the assignment operator, `=`, associate from left to right. Addition is left associative, so an expression like \( x + y + z \) is evaluated as if it had been written as \((x + y) + z\). The assignment operator, `=`, associates from right to left, so an expression like \( x = y = 0 \) is evaluated as if it had been written as \( x = (y = 0) \), which, as we will soon see, first assigns the value 0 to variable \( y \) and then assigns the result of that assignment, 0, to \( x \).

Good Programming Practice 2.20

Refer to the operator precedence chart (see the complete chart in Appendix C) when writing expressions containing many operators. Confirm that the operations in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, use parentheses to force the order, exactly as you would do in algebraic expressions. Be sure to observe that some operators, such as assignment, `=`, associate from right to left rather than from left to right.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
<td>parentheses</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

Fig. 2.21 Precedence and associativity of the operators discussed so far.
We have introduced many important features of Java in this chapter, including displaying data on the screen, inputting data from the keyboard, performing calculations and making decisions. We should note that these applications are meant to introduce the reader to basic programming concepts. As you will see in later chapters, more substantial Java applications contain just a few lines of code in method `main` that creates the objects that perform the work of the application. In Chapter 3, we demonstrate many similar techniques as we introduce Java applet programming. In Chapter 4, we build on the techniques of Chapter 2 and Chapter 3 as we introduce structured programming. You will become more familiar with indentation techniques. We will study how to specify and vary the order in which statements are executed; this order is called flow of control.

2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

Now we begin our optional, object-oriented design and implementation case study. The “Thinking About Objects” sections at the ends of this and the next several chapters will ease you into object orientation by examining an elevator simulation case study. This case study will provide you with a substantial, carefully paced, complete design and implementation experience. In Chapters 3 through 13, Chapter 15 and Chapter 22, we will perform the various steps of an object-oriented design (OOD) process using the UML while relating to the object-oriented concepts discussed in the chapters. In Appendices G, H and I, we will implement the elevator simulator using the techniques of object-oriented programming (OOP) in Java. We present the complete case-study solution. This is not an exercise; rather, it is an end-to-end learning experience that concludes with a detailed walkthrough of the actual Java code. We have provided this case study so that you can become accustomed to the kinds of substantial problems encountered in industry. We hope you enjoy this learning experience.

**Problem Statement**

A company wants you to develop an object-oriented software-simulator application in Java that models the operation of the elevator to determine whether it will meet the company’s needs. The company wants the simulation to contain an elevator system. The application consists of three parts. The first and most substantial part is the simulator, which models the operation of the elevator system. The second part is the display of this model on screen so that the user may view it graphically. The final part is the graphical user interface, or GUI, that allows the user to control the simulation. Our design and implementation will follow the so-called Model-View-Controller architecture we will learn about in Section 13.17.

The elevator system consists of an elevator shaft and an elevator car. In our simulation, we model people who ride the elevator car (referred to as “the elevator”) to travel between the floors in the elevator shaft, as shown in Fig. 2.22, Fig. 2.23 and Fig. 2.24.

The elevator contains a door (called the “elevator door”) that opens upon the elevator’s arrival at a floor and closes upon the elevator’s departure from that floor. The elevator door is closed during the trips between floors to prevent the passenger from being injured by brushing against the wall of the elevator shaft. In addition, the elevator shaft connects to a door on each floor (referred to as the two “floor doors”), so people cannot fall down the shaft when the elevator is not at a floor. Note that we do not display the floor doors in the figures, because they would obscure the inside of the elevator (we use a mesh door to rep-
resent the elevator door because mesh allows us to see inside the elevator). The floor door opens concurrently with the elevator door, so it appears as if both doors open at the same time. A person sees only one door, depending on that person’s location. When the person is inside the elevator, the person sees the elevator door and can exit the elevator when this door opens; when the person is outside the elevator, the person sees the floor door and can enter the elevator when that door opens.\(^1\)

The elevator starts on the first floor with all the doors closed. To conserve energy, the elevator moves only when necessary. For simplicity, the elevator and floors each have a capacity of only one person.\(^2\)

The user of our application should, at any time, be able to create a unique person in the simulation and situate that person on either the first or second floor (Fig. 2.22). When created, the person walks across the floor to the elevator. The person then presses a button on the floor next to the elevator shaft (referred to as a “floor button”). When pressed, that floor button illuminates, then requests the elevator. When summoned, the elevator travels to the person’s floor. If the elevator is already on that person’s floor, the elevator does not travel. Upon arrival, the elevator resets the button inside the elevator (called the “elevator button”), sounds the bell inside the elevator, then opens the elevator door (which opens the floor door on that floor). The elevator then signals the elevator shaft of the arrival. The elevator shaft, upon receiving this message, resets the floor button and illuminates the light on that floor.

Occasionally, a person requests the elevator when it is moving. If the request was generated at the floor from which the elevator just departed, the elevator must “remember” to revisit that floor after carrying the current passenger to the other floor.

When the floor door opens, the person enters the elevator after the elevator passenger (if there is one) exits. If a person neither enters nor requests the elevator, the elevator closes its door and remains on that floor until the next person presses a floor button to summon the elevator.

When a person enters the elevator, that person presses the elevator button, which also illuminates when pressed. The elevator closes its door (which also closes the floor door on that floor) and moves to the opposite floor. The elevator takes five seconds to travel between floors. When the elevator arrives at the destination floor, the elevator door opens (along with the floor door on that floor) and the person exits the elevator.

The application user introduces a person onto the first or second floor by pressing the First Floor button or the Second Floor button, respectively. When the user presses the First Floor button, a person should be created (by the elevator simulation) and positioned on the first floor of the building. When the user presses the Second Floor button, a person should be created and positioned on the second floor. Over time, the user can create any number of people in the simulation, but the user cannot create a new person on an occupied floor. For example, Fig. 2.22 shows that the First Floor button is disabled to prevent the user from creating more than one person on the first floor. Figure 2.23 shows that this button is reenabled when the person rides the elevator.

---

1. Most people do not consider this when riding an elevator—they really think of one “elevator door,” when in reality, there is a door in the elevator and a door on the floor, and these doors open and close in tandem.
2. After you have studied this case study, you may want to modify it to allow more than one person to ride the elevator at once and more than one person to wait on each floor at once.
Fig. 2.22  Person moving towards elevator on the first floor.

Fig. 2.23  Person riding the elevator to the second floor.
The company requests that we display the results of the simulation graphically, as shown in Fig. 2.22, Fig. 2.23 and Fig. 2.24. At various points in time, the screen should display a person walking to the elevator, pressing a button and entering, riding and exiting the elevator. The display also should also show the elevator moving, the doors opening, the lights turning on and off, the buttons illuminating when they are pressed and the buttons darkening when they are reset.

The company requests that audio be integrated into the simulation. For example, as a person walks, the application user should hear the footsteps. Each time a floor or elevator button is pressed or reset, the user should hear a click. The bell should ring upon the elevator’s arrival, and doors should creak when they open or close. Lastly, “elevator music” should play as the elevator travels between floors.

**Analyzing and Designing the Elevator System**

We must analyze and design our system before we implement it as Java code. The output of the analysis phase is intended to specify clearly in a *requirements document* what the system is supposed to do. The requirements document for this case study is essentially the description of what the elevator simulator is supposed to do—presented informally in the last few pages. The output of the design phase should specify clearly *how* the system should be constructed to do what is needed. In the next several “Thinking About Objects” sections, we perform the steps of an object-oriented design (OOD) process on the elevator system. The UML is designed for use with any OOD process—many such processes exist. One popular method is the Rational Unified Process™ developed by Rational Software Corporation. For this case study, we present our own simplified design process. For many of our readers, this will be their first OOD/UML experience.

---

**Fig. 2.24** Person walking away from elevator.
We now begin the design phase of our elevator system, which will span Chapters 2 through 13, Chapter 15 and Chapter 22, in which we gradually develop the design. Appendices G, H and I present the complete Java implementation.

A system is a set of components that interact to solve a problem. In our case study, the elevator-simulator application represents the system. A system may contain “subsystems,” which are “systems within a system.” Subsystems simplify the design process by managing subsets of system responsibilities. System designers may allocate system responsibilities among the subsystems, design the subsystems, then integrate the subsystems with the overall system. Our elevator-simulation system contains three subsystems, which are defined in the problem statement:

1. the simulator model (which represents the operation of the elevator system),
2. the display of this model on screen (so that the user may view it graphically), and
3. the graphical user interface (that allows the user to control the simulation).

We develop the simulator model gradually through Chapter 15 and present the implemented model in Appendix H. We discuss the GUI components allowing the user to control the model in Chapter 12 and introduce how the subsystems work together to form the system in Chapter 13. Finally, we introduce how to display the simulator model on the screen in Chapter 22 and conclude the display in Appendix I.

System structure describes the system’s objects and their inter-relationships. System behavior describes how the system changes as its objects interact with each other. Every system has both structure and behavior—we must design both. However, there are several distinct types of system structures and behaviors. For example, the interaction among the objects in the system differs from the interaction between the user and the system, yet both are interactions that constitute the system behavior.

The UML specifies nine types of diagrams for modeling systems. Each diagram models a distinct characteristic of a system’s structure or behavior—the first four diagrams relate to system structure; the remaining five diagrams relate to system behavior:

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram
5. Activity diagram
6. Statechart diagram
7. Collaboration diagram
8. Sequence diagram
9. Use-Case diagram

Class diagrams, which we explain in “Thinking About Objects” Section 3.8, model the classes, or “building blocks,” used to build a system. Each entity in the problem statement is a candidate to be a class in the system (i.e., Person, Elevator, Floor, etc.).

Object diagrams, which we also explain in Section 3.8, model a “snapshot” of the system by modeling a system’s objects and their relationships at a specific point in time. Each object represents an instance of a class from the class diagram (e.g., the elevator
object is an instance of class **Elevator**), and there may be several objects created from one class (e.g., both the first floor button object and the second floor button object are created from class **FloorButton**).

*Component diagrams*, presented in Section 13.17, model the *components*—resources (which include graphics and audio) and *packages* (which are groups of classes)—that make up the system.

*Deployment diagrams* model the runtime requirements of the system (such as the computer or computers on which the system will reside), memory requirements for the system, or other devices the system requires during execution. We do not present deployment diagrams in this case study, because we are not designing a “hardware-specific” system—our simulation requires only one computer containing the Java 2 runtime environment on which to run.

*Statechart diagrams*, which we introduce in Section 5.11, model how an object changes *state* (i.e., the condition of an object at a specific time). When an object changes state, that object may behave differently in the system.

*Activity diagrams*, which we also introduce in Section 5.11, model an object’s *activity*—that object’s workflow during program execution. An activity diagram is a flowchart that models the actions the object will perform and in what order.

Both *collaboration diagrams* and *sequence diagrams* model the interactions among the objects in a system. Collaboration diagrams emphasize what interactions occur, whereas sequence diagrams emphasize when interactions occur. We introduce these diagrams in Section 7.10 and Section 15.12, respectively.

*Use-Case diagrams* represent the interaction between the user and our system (i.e., all actions the user may perform on the system). We introduce use-case diagrams in Section 12.16, where we discuss user-interface issues.

In “Thinking About Objects” Section 3.17, we continue designing our elevator system by identifying the classes in the problem statement. We accomplish this by extracting all the nouns and noun clauses from the problem statement. Using these classes, we develop a class diagram that models the structure of our elevator simulation system.

**Internet and World-Wide-Web Resources**

Listed below are URLs and books on object-oriented design with the UML—you may find these references helpful as you study the remaining sections of our case-study presentation.

  This is the UML resource page from the Object Management Group, which provides specifications for various object-oriented technologies, such as the UML.

  This site shows how to draw UML diagrams without the use of modeling tools.

- [www.rational.com/uml/index.jsp](http://www.rational.com/uml/index.jsp)
  This is the UML resource page for Rational Software Corporation—the company that created the UML.

- [microgold.com/Stage/UML_FAQ.html](http://microgold.com/Stage/UML_FAQ.html)
  This site provides the UML FAQ maintained by Rational Software.

  This site hosts the Unified Modeling Language Dictionary, which lists and defines all terms used in the UML.
www.embarcadero.com
This site provides a free 30-day license to download a trial-version of Describe™ — the new UML modeling tool from Embarcadero Technologies®.

www.ics.uci.edu/pub/arch/uml/uml_books_and_tools.html
This site lists books on the UML and software tools that use the UML, such as Rational Rose™ and Embarcadero Describe™.

www.ootips.org/ood-principles.html
This site provides answers to the question “what makes good OOD?”

wdvl.internet.com/Authoring/Scripting/Tutorial/oo_design.html
This site introduces OOD and provides OOD resources.

Bibliography


SUMMARY
• An application is a program that executes using the java interpreter.
• A comment that begins with // is called a single-line comment. Programmers insert comments to document programs and improve program readability.
• A string of characters contained between double quotation marks is called a string, a character string, a message or a string literal.
• Blank lines, space characters, newline characters and tab characters are known as white-space characters. White-space characters outside strings are ignored by the compiler.
Keyword **class** introduces a class definition and is immediately followed by the class name.

Keywords (or reserved words) are reserved for use by Java. Keywords must appear in all lower-case letters.

By convention, all class names in Java begin with a capital letter. If a class name contains more than one word, the first letter of each word should be capitalized.

An identifier is a series of characters consisting of letters, digits, underscores ( _ ) and dollar signs ( $ ) that does not begin with a digit, does not contain any spaces and is not a keyword.

Java is case sensitive—that is, uppercase and lowercase letters are different.

A left brace, {, begins the body of every class definition. A corresponding right brace, }, ends each class definition.

Java applications begin executing at method **main**.

Methods are able to perform tasks and return information when they complete their tasks.

The first line of method **main** must be defined as

```
public static void main( String args[] )
```

A left brace, {, begins the body of a method definition. A corresponding right brace, }, ends the method definition’s body.

**System.out** is known as the standard output object. **System.out** allows Java applications to display strings and other types of information in the command window from which the Java application executes.

The escape sequence \n indicates a newline character. Other escape sequences include \t (tab), \r (carriage return), \ (backslash) and " (double quote).

Method **println** of the **System.out** object displays (or prints) a line of information in the command window. When **println** completes its task, it automatically positions the output cursor to the beginning of the next line in the command window.

Every statement must end with a semicolon (also known as the statement terminator).

The difference between **System.out**’s **print** and **println** methods is that **print** does not position to the beginning of the next line in the command window when it finishes displaying its argument. The next character displayed in the command window appears immediately after the last character displayed with **print**.

Java contains many predefined classes that are grouped into categories of related classes called packages. The packages are referred to collectively as the Java class library or the Java applications programming interface (Java API).

Class **JOptionPane** is defined in package **javax.swing**. Class **JOptionPane** contains methods that display dialog boxes.

The compiler uses **import** statements to locate classes required to compile a Java program.

The **javax.swing** package contains many classes that help define a graphical user interface (GUI) for an application. GUI components facilitate data entry by the user of a program and data outputs by a program.

Method **showMessageDialog** of class **JOptionPane** displays a dialog box containing a message to the user.

A **static** method is called by following its class name by a dot ( . ) and the name of the method.

Method **exit** of class **System** terminates an application. Class **System** is in package **java.lang**. All Java programs import package **java.lang** by default.
• A variable is a location in the computer’s memory where a value can be stored for use by a program. The name of a variable is any valid identifier.
• All variables must be declared with a name and a data type before they can be used in a program.
• Declarations end with a semicolon (;) and can be split over several lines, with each variable in the declaration separated by a comma (forming a comma-separated list of variable names).
• Variables of type `int` hold integer values (whole numbers such as 7, –11, 0 and 31,914).
• Types such as `int`, `float`, `double` and `char` are primitive data types. Names of primitive data types are keywords of the Java programming language.
• A prompt directs the user to take a specific action.
• A variable is assigned a value by using an assignment statement, which uses the assignment operator, `.=`. The `=` operator is called a binary operator, because it has two operands.
• Method `Integer.parseInt` (a `static` method of class `Integer`) converts its `String` argument to an `int` value.
• Java has a version of the `+` operator for string concatenation that enables a string and a value of another data type (including another string) to be concatenated.
• Every variable has a name, a type, a size and a value.
• When a value is placed in a memory location, the value replaces the value previously in that location. When a value is read out of a memory location, the variable’s value remains unchanged.
• The arithmetic operators are binary operators, because they operate on two operands.
• Integer division yields an integer result.
• Arithmetic expressions in Java must be written in straight-line form to facilitate entering programs into the computer.
• Operators in arithmetic expressions are applied in a precise sequence determined by the rules of operator precedence.
• Parentheses may be used to force the order of evaluation of operators.
• When we say that operators are applied from left to right, we are referring to the associativity of the operators. Some operators associate from right to left.
• Java’s `if` structure allows a program to make a decision based on the truth or falsity of a condition. If the condition is met (i.e., the condition is true), the statement in the body of the `if` structure executes. If the condition is not met (i.e., the condition is false), the body statement does not execute.
• Conditions in `if` structures can be formed by using the equality operators and relational operators.
• The empty string is a string containing no characters.
• Every variable declared in a method must be initialized before it can be used in an expression.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Addition operator (+)</th>
<th>Body of a class definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applet</td>
<td>Body of a method definition</td>
</tr>
<tr>
<td>Application</td>
<td>Braces ({ and })</td>
</tr>
<tr>
<td>Argument to a method</td>
<td>Case sensitive</td>
</tr>
<tr>
<td>Arithmetic operators</td>
<td>Character string</td>
</tr>
<tr>
<td>Assignment operator (=)</td>
<td>Class</td>
</tr>
<tr>
<td>Assignment statement</td>
<td>Class definition</td>
</tr>
<tr>
<td>Associativity of operators</td>
<td>.class file extension</td>
</tr>
<tr>
<td>Backslash () escape character</td>
<td>Class keyword</td>
</tr>
<tr>
<td>Binary operator</td>
<td>Class name</td>
</tr>
</tbody>
</table>
command tool
command window
comma-separated list
comment (/ /)
compilation error
compile error
compiler
compile-time error
condition
decision
declaration
dialog
dialog box
division operator (/)
document a program
empty string ("")
equality operators
    == “is equal to”
    != “is not equal to”
escape sequence
exit method of System
false
graphical user interface (GUI)
identifier
if structure
import statement
input dialog
int primitive type
integer (int)
Integer class
integer division
interpreter
Java
Java 2 Software Development Kit (J2SDK)
Java applications programming interface (API)
Java class library
Java documentation comment
.java file extension
java interpreter
java.lang package
javax.swing package
JOptionPane class
JOptionPane.ERROR_MESSAGE
JOptionPane.INFORMATION_MESSAGE
JOptionPane.QUESTION_MESSAGE
JOptionPane.WARNING_MESSAGE
left brace, {, begins the body of a class
left brace, {, begins the body of a method
literal
main method
memory
memory location
message
message dialog
method
Microsoft Internet Explorer browser
modulus operator (%)
mouse cursor
mouse pointer
MS-DOS Prompt
multiple-line comment
multiplication operator (*)
nested parentheses
Netscape Navigator browser
newline character (\n)
object
operand
operator
package
parentheses ()
parseInt method of class Integer
precedence
primitive data type
 programmer-defined class
prompt
public keyword
relational operators
    < “is less than”
    <= “is less than or equal to”
    > “is greater than”
    >= “is greater than or equal to”
reserved words
right brace, }, ends the body of a class
right brace, }, ends the body of a method
right-to-left associativity
rules of operator precedence
semicolon (;) statement terminator
shell tool
showInputDialog method of JOptionPane
showMessageDialog method of
    JOptionPane

static method
straight-line form
string
String class
SELF-REVIEW EXERCISES

2.1 Fill in the blanks in each of the following statements:
   a) The _________ begins the body of every method, and the _________ ends the body of every method.
   b) Every statement ends with a _________.
   c) The _________ structure is used to make decisions.
   d) _________ begins a single-line comment.
   e) _________, _________, _________ and _________ are called white-space characters.
   f) Class _________ contains methods that display message dialogs and input dialogs.
   g) _________ are reserved for use by Java.
   h) Java applications begin execution at method _________.
   i) Methods _________ and _________ display information in the command window.
   j) A _________ method is always called using its class name followed by a dot (.) and its method name.

2.2 State whether each of the following is true or false. If false, explain why.
   a) Comments cause the computer to print the text after the // on the screen when the program is executed.
   b) All variables must be given a type when they are declared.
   c) Java considers the variables number and NuMbEr to be identical.
   d) The modulus operator (%) can be used only with integer operands.
   e) The arithmetic operators *, /, %, + and - all have the same level of precedence.
   f) Method Integer.parseInt converts an integer to a String.

2.3 Write Java statements to accomplish each of the following tasks:
   a) Declare variables c, thisIsAVariable, q76354 and number to be of type int.
   b) Display a dialog asking the user to enter an integer.
   c) Convert a String to an integer, and store the converted value in integer variable age. Assume that the String is stored in value.
   d) If the variable number is not equal to 7, display "The variable number is not equal to 7" in a message dialog. [Hint: Use the version of the message dialog that requires two arguments.]
   e) Print the message "This is a Java program" on one line in the command window.
   f) Print the message "This is a Java program" on two lines in the command window; the first line should end with Java. Use only one statement.

2.4 Identify and correct the errors in each of the following statements:
   a) if ( c < 7 );
      JOptionPane.showMessageDialog( null, "c is less than 7");
   b) if ( c => 7 )
      JOptionPane.showMessageDialog( null, "c is equal to or greater than 7");

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
2.5 Write a statement (or comment) to accomplish each of the following tasks:
   a) State that a program will calculate the product of three integers.
   b) Declare the variables \( x, y, z \) and \( \text{result} \) to be of type \( \text{int} \).
   c) Declare the variables \( \text{xVal, yVal} \) and \( \text{zVal} \) to be of type \( \text{String} \).
   d) Prompt the user to enter the first value, read the value from the user and store it in the variable \( \text{xVal} \).
   e) Prompt the user to enter the second value, read the value from the user and store it in the variable \( \text{yVal} \).
   f) Prompt the user to enter the third value, read the value from the user and store it in the variable \( \text{zVal} \).
   g) Convert \( \text{xVal} \) to an \( \text{int} \), and store the result in the variable \( x \).
   h) Convert \( \text{yVal} \) to an \( \text{int} \), and store the result in the variable \( y \).
   i) Convert \( \text{zVal} \) to an \( \text{int} \), and store the result in the variable \( z \).
   j) Compute the product of the three integers contained in variables \( x, y \) and \( z \), and assign the result to the variable \( \text{result} \).
   k) Display a dialog containing the message "The product is " followed by the value of the variable \( \text{result} \).
   l) Return a value from the program indicating that the program terminated successfully.

2.6 Using the statements you wrote in Exercise 2.5, write a complete program that calculates and prints the product of three integers.

ANSWERS TO SELF-REVIEW EXERCISES

2.1 a) left brace (\{), right brace (\}). b) semicolon (;). c) \text{if}. d) \text{/}. e) Blank lines, space characters, newline characters and tab characters. f) \text{JOptionPane}. g) Keywords. h) \text{main}. i) \text{System.out.print} and \text{System.out.println}. j) \text{static}.

2.2 a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs and improve their readability.
   b) True.
   c) False. Java is case sensitive, so these variables are distinct.
   d) False. The modulus operator can also be used with noninteger operands in Java.
   e) False. The operators \( \ast, / \) and \( \% \) are on the same level of precedence, and the operators \( + \) and \( - \) are on a lower level of precedence.
   f) False. \text{Integer.parseInt} method converts a \text{String} to an integer (\text{int}) value.

2.3 a) \text{int c, thisIsAVariable, q76354, number;}
   b) \text{value = JOptionPane.showInputDialog( "Enter an integer" );}
   c) \text{age = Integer.parseInt( value );}
   d) \text{if ( number != 7 )}
      \text{JOptionPane.showMessageDialog( null,}
      \text{"The variable number is not equal to 7" );}
   e) \text{System.out.println( "This is a Java program" );}
   f) \text{System.out.println( "This is a Java
program" );}

2.4 The solutions to Self-Review Exercise 2.4 are as follows:
   a) Error: Semicolon after the right parenthesis of the condition in the \text{if} statement.
      Correction: Remove the semicolon after the right parenthesis. \( \text{[Note: The result of this}\)
      error is that the output statement will be executed regardless of whether the condition in
      the \text{if} statement is true. The semicolon after the right parenthesis is considered an empty
      statement—a statement that does nothing. We will learn more about the empty statement
      in the next chapter.\)
b) Error: The relational operator => is incorrect. Correction: Change => to >=.

2.5

a) // Calculate the product of three integers
b) int x, y, z, result;
c) String xVal, yVal, zVal;
d) xVal = JOptionPane.showInputDialog(
   "Enter first integer:"
);
e) yVal = JOptionPane.showInputDialog(
   "Enter second integer:"
);
f) zVal = JOptionPane.showInputDialog(
   "Enter third integer:"
);
g) x = Integer.parseInt( xVal );
h) y = Integer.parseInt( yVal );
i) z = Integer.parseInt( zVal );
j) result = x * y * z;
k) JOptionPane.showMessageDialog( null,
   "The product is " + result );
l) System.exit( 0 );

2.6 The solution to Exercise 2.6 is as follows:

```java
// Calculate the product of three integers

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

public class Product {

   public static void main( String args[] )
   {
      int x, y, z, result;
      String xVal, yVal, zVal;

      xVal = JOptionPane.showInputDialog(
         "Enter first integer:"
      );
      yVal = JOptionPane.showInputDialog(
         "Enter second integer:"
      );
      zVal = JOptionPane.showInputDialog(
         "Enter third integer:"
      );
      x = Integer.parseInt( xVal );
      y = Integer.parseInt( yVal );
      z = Integer.parseInt( zVal );
      result = x * y * z;
      JOptionPane.showMessageDialog( null,
         "The product is " + result );
      System.exit( 0 );
   }
}
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
EXERCISES

2.7 Fill in the blanks in each of the following statements:
a) ________ are used to document a program and improve its readability.
b) An input dialog capable of receiving input from the user is displayed with method ________ of class ________.
c) A decision can be made in a Java program with an ________.
d) Calculations are normally performed by ________ statements.
e) A dialog capable of displaying a message to the user is displayed with method ________ of class ________.

2.8 Write Java statements that accomplish each of the following tasks:
a) Display the message "Enter two numbers", using class JOptionPane.
b) Assign the product of variables b and c to variable a.
c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).

2.9 State whether each of the following is true or false. If false, explain why.
a) Java operators are evaluated from left to right.
b) The following are all valid variable names: _under_bar_, m928134, t5, j7, her_sales$, his_$account_total, a, b$, c, z, z2.
c) A valid Java arithmetic expression with no parentheses is evaluated from left to right.
d) The following are all invalid variable names: 3g, 87, 67h2, h22, 2h.

2.10 Fill in the blanks in each of the following statements:
a) What arithmetic operations have the same precedence as multiplication? ________.
b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? ________.
c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a ________.

2.11 What displays in the message dialog when each of the given Java statements is performed? Assume that \( x = 2 \) and \( y = 3 \).
a) JOptionPane.showMessageDialog( null, "x = " + x);
b) JOptionPane.showMessageDialog( null, "The value of x + x is " + ( x + x ));
c) JOptionPane.showMessageDialog( null, "x =" );
d) JOptionPane.showMessageDialog( null, ( x + y ) + " = " + ( y + x ));
2.12 Which of the following Java statements contain variables whose values are changed or replaced?
   a) \( p = i + j + k + 7; \)
   b) `JOptionPane.showMessageDialog(null, "variables whose values are destroyed");`
   c) `JOptionPane.showMessageDialog(null, "a = 5");`
   d) `stringVal = JOptionPane.showInputDialog("Enter string:");`

2.13 Given that \( y = ax^3 + 7 \), which of the following are correct Java statements for this equation?
   a) \( y = a \times x \times x \times x + 7; \)
   b) \( y = (a \times x) \times x \times (x + 7); \)
   c) \( y = a \times (x \times x \times x) + 7; \)
   d) \( y = a \times x \times (x \times x + 7); \)

2.14 State the order of evaluation of the operators in each of the following Java statements, and show the value of \( x \) after each statement is performed:
   a) \( x = 7 + 3 \times 6 / 2 - 1; \)
   b) \( x = 2 \% 2 + 2 \times 2 - 2 / 2; \)
   c) \( x = (3 \times 9 \times (3 + (9 \times 3 / (3))) )); \)

2.15 Write an application that displays the numbers 1 to 4 on the same line, with each pair of adjacent numbers separated by one space. Write the program using the following methods:
   a) Using one `System.out` statement.
   b) Using four `System.out` statements.

2.16 Write an application that asks the user to enter two numbers, obtains the numbers from the user and prints the sum, product, difference and quotient (division) of the numbers. Use the techniques shown in Fig. 2.9.

2.17 Write an application that asks the user to enter two integers, obtains the numbers from the user and displays the larger number followed by the words “is larger” in an information message dialog. If the numbers are equal, print the message “These numbers are equal.” Use the techniques shown in Fig. 2.20.

2.18 Write an application that inputs three integers from the user and displays the sum, average, product, smallest and largest of the numbers in an information message dialog. Use the GUI techniques shown in Fig. 2.20. [Note: The calculation of the average in this exercise should result in an integer representation of the average. So, if the sum of the values is 7, the average should be 2, not 2.3333…]

2.19 Write an application that inputs from the user the radius of a circle and prints the circle’s diameter, circumference and area. Use the value 3.14159 for \( \pi \). Use the GUI techniques shown in Fig. 2.9. [Note: You may also use the predefined constant `Math.PI` for the value of \( \pi \). This constant is more precise than the value 3.14159. Class `Math` is defined in the `java.lang` package, so you do not need to `import` it.] Use the following formulas (\( r \) is the radius):

\[
diameter = 2r \\
circumference = 2\pi r \\
area = \pi r^2
\]

2.20 Write an application that displays in the command window a box, an oval, an arrow and a diamond using asterisks (*), as follows:
2.21 Modify the program you created in Exercise 2.20 to display the shapes in a `JOptionPane.PLAIN_MESSAGE` dialog. Does the program display the shapes exactly as in Exercise 2.20?

2.22 What does the following code print?

```java
System.out.println("\n***\n*****\n******\n******");
```

2.23 What does the following code print?

```java
System.out.println("**");
System.out.println("***");
System.out.println("*****");
System.out.println("****");
System.out.println("**");
```

2.24 What does the following code print?

```java
System.out.print("**");
System.out.print("***");
System.out.print("*****");
System.out.print("****");
System.out.println("**");
```

2.25 What does the following code print?

```java
System.out.print("**");
System.out.println("***");
System.out.println("*****");
System.out.println("****");
System.out.println("**");
```

2.26 Write an application that reads five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.27 Write an application that reads an integer and determines and prints whether it is odd or even. [Hint: Use the modulus operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of 0 when divided by 2.]

2.28 Write an application that reads in two integers and determines and prints if the first is a multiple of the second. [Hint: Use the modulus operator.]

2.29 Write an application that displays in the command window a checkerboard pattern as follows:
2.30 Modify the program you wrote in Exercise 2.29 to display the checkerboard pattern in a `JOptionPane.PLAIN_MESSAGE` dialog. Does the program display the shapes exactly as in Exercise 2.29?

2.31 Here’s a peek ahead. In this chapter, you have learned about integers and the data type `int`. Java can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. Every character has a corresponding integer representation. The set of characters a computer uses and the corresponding integer representations for those characters is called that computer’s *character set*. You can indicate a character value in a program simply by enclosing that character in single quotes, as in `'A'`.

You can determine the integer equivalent of a character by preceding that character with `(int)`. This form is called a cast (we will say more about casts in Chapter 4) as in:

```
(int) 'A'
```

The following statement outputs a character and its integer equivalent:

```
System.out.println( "The character " + 'A' + 
    " has the value " + ( int ) 'A' );
```

When the preceding statement executes, it displays the character `A` and the value `65` (from the so-called Unicode character set) as part of the string.

Write an application that displays the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. At a minimum, display the integer equivalents of the following: A B C a b c 0 1 2 $ * + / and the blank character.

2.32 Write an application that inputs one number consisting of five digits from the user, separates the number into its individual digits and prints the digits separated from one another by three spaces each. For example, if the user types in the number 42339, the program should print

```
4   2   3   3   9
```

*[Hint: It is possible to do this exercise with the techniques you learned in this chapter. You will need to use both division and modulus operations to “pick off” each digit.]*

Assume that the user enters the correct number of digits. What happens when you execute the program and type a number with more than five digits? What happens when you execute the program and type a number with fewer than five digits?
2.33 Using only the programming techniques you learned in this chapter, write an application that calculates the squares and cubes of the numbers from 0 to 10 and prints the resulting values in table format as follows:

<table>
<thead>
<tr>
<th>number</th>
<th>square</th>
<th>cube</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>216</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>343</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>729</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
</tbody>
</table>

[Note: This program does not require any input from the user.]

2.34 Write a program that reads a first name and a last name from the user as two separate inputs and concatenates the first name and last name, separating them by a space. Display in a message dialog the concatenated name.

2.35 Write a program that inputs five numbers and determines and prints the number of negative numbers input, the number of positive numbers input and the number of zeros input.
Introduction to Java Applets

Objectives

• To observe some of Java’s exciting capabilities through the Java 2 Software Development Kit’s demonstration applets.
• To differentiate between applets and applications.
• To be able to write simple Java applets.
• To be able to write simple Hypertext Markup Language (HTML) files to load an applet into the appletviewer or a World Wide Web browser.
• To understand the difference between variables and references.
• To execute applets in World Wide Web browsers.

He would answer to “Hi!” or to any loud cry
Such as “Fry me!” or “Fritter my wig!”
To “What-you-may-call-um!” or “What-was-his-name!”
But especially “Thing-um-a-jig!”
Lewis Carroll

Painting is only a bridge linking the painter's mind with that
of the viewer.
Eugène Delacroix

My method is to take the utmost trouble to find the right thing
to say, and then to say it with the utmost levity.
George Bernard Shaw

Though this be madness, yet there is method in ’t.
William Shakespeare
3.1 Introduction

In Chapter 2, we introduced Java application programming and several important aspects of Java applications. This chapter introduces another type of Java program called a Java applet. Applets are Java programs that can be embedded in Hypertext Markup Language (HTML) documents (i.e., Web pages). When a browser loads a Web page containing an applet, the applet downloads into the Web browser and begins execution.

The browser that executes an applet is generically known as the applet container. The Java 2 Software Development Kit (J2SDK) includes an applet container (called the appletviewer) for testing applets before you embed them in a Web page. Most Web browsers in use today do not support Java 2 directly. For this reason, we normally demonstrate our applets using the appletviewer. One browser that does support Java 2 is Netscape Navigator 6. To execute applets in other Web browsers such as Microsoft Internet Explorer or earlier versions of Netscape Navigator requires the Java Plug-in, which we discuss in Section 3.6.2 of this chapter.

Portability Tip 3.1

Most Web browsers in use today do not support applets written in Java 2. To execute applets in such browsers, you must use the Java Plug-in (see Section 3.6.2).

Testing and Debugging Tip 3.1

Test your applets in the appletviewer applet container before executing them in a Web browser. This enables you to see error messages that may occur. Also, once an applet is executing in a browser, it is sometimes difficult to reload the applet after making changes to the applet’s class definition.
Testing and Debugging Tip 3.2

Test your applets in every Web browser in which the applets will execute to ensure that they operate correctly in each browser.

One of our goals in this chapter is to mimic several features presented in Chapter 2. This provides positive reinforcement of previous concepts. Another goal of this chapter is to begin using the object-oriented programming terminology introduced in Section 1.15.

As in Chapter 2, there are a few cases where we do not as yet provide all the details necessary to create complex applications and applets in Java. It is important to build your knowledge of fundamental programming concepts first. In Chapter 4 and Chapter 5, we present a detailed treatment of program development and program control in Java. As we proceed through the text, we present many substantial applications and applets.

3.2 Sample Applets from the Java 2 Software Development Kit

We begin by considering several sample applets provided with the Java 2 Software Development Kit (J2SDK) version 1.3. The applets we demonstrate give you a sense of Java’s capabilities. Each of the sample programs provided with the J2SDK also comes with source code (the .java files containing the Java applet programs). This source code is helpful as you enhance your Java knowledge—you can read the source code provided to learn new and exciting features of Java. Remember, all programmers initially learn new features by mimicking their use in existing programs. The J2SDK comes with many such programs and there are a tremendous number of Java resources on the Internet and World Wide Web that include Java source code.

The demonstration programs provided with the J2SDK are located in your J2SDK install directory in a subdirectory called demo. For the Java 2 Software Development Kit version 1.3, the default location of the demo directory on Windows is

```
c:\jdk1.3\demo
```

On UNIX/Linux it is the directory in which you install the J2SDK followed by jdk1.3/demo—for example

```
/usr/local/jdk1.3/demo
```

For other platforms, there will be a similar directory (or folder) structure. For the purpose of this chapter, we assume on Windows that the J2SDK is installed in c:\jdk1.3 and on UNIX that the J2SDK is installed in your home directory in ~/jdk1.3. [Note: You may need to update these locations to reflect your chosen install directory and/or disk drive, or a newer version of the J2SDK.]

If you are using a Java Development tool that does not come with the Sun Java demos, you can download the J2SDK (with the demos) from the Sun Microsystems Java Web site

```
java.sun.com/j2se/1.3/
```

3.2.1 The TicTacToe Applet

The first applet we demonstrate from the J2SDK demos is the TicTacToe applet, which allows you to play Tic-Tac-Toe against the computer. To execute this applet, open a com-
mand window (MS-DOS Prompt on Windows 95/98/ME, Command Prompt on Windows NT/2000 or a command tool/shell tool on UNIX) and change directories to the J2SDK’s demo directory. Both Windows and UNIX use command cd to change directories. For example, the command

```
cd c:\jdk1.3\demo
```
changes to the demo directory on Windows and the command

```
cd ~/jdk1.3/demo
```
changes to the demo directory on UNIX.

The demo directory contains four subdirectories—applets, jfc, jpda and sound (you can see these directories by issuing in the command window the dir command on Windows or the ls command on UNIX). The applets directory contains many demonstration applets. The jfc (Java Foundation Classes) directory contains many examples of Java’s newest graphics and GUI features (some of these examples are also applets). The jdpa directory contains examples of the Java Platform Debugging Architecture (beyond the scope of this book). The sound directory contains examples of the Java Sound API (covered in Chapter 18). For the demonstrations in this section, change directories to the applets directory by issuing the command

```
cd applets
```
on either Windows or UNIX.

Listing the contents of the applets directory (with the dir command on Windows or the ls command on UNIX) indicates that there are many examples. Figure 3.1 shows the subdirectories and provides a brief description of the examples in each subdirectory.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animator</td>
<td>Performs one of four separate animations.</td>
</tr>
<tr>
<td>ArcTest</td>
<td>Demonstrates drawing arcs. You can interact with the applet to change</td>
</tr>
<tr>
<td></td>
<td>attributes of the arc that is displayed.</td>
</tr>
<tr>
<td>BarChart</td>
<td>Draws a simple bar chart.</td>
</tr>
<tr>
<td>Blink</td>
<td>Displays blinking text in different colors.</td>
</tr>
<tr>
<td>CardTest</td>
<td>Demonstrates several GUI components and a variety of ways in which GUI</td>
</tr>
<tr>
<td></td>
<td>components can be arranged on the screen (the arrangement of GUI components</td>
</tr>
<tr>
<td></td>
<td>is also known as the layout of the GUI components).</td>
</tr>
<tr>
<td>Clock</td>
<td>Draws a clock with rotating “hands,” the current date and the current time.</td>
</tr>
<tr>
<td></td>
<td>The clock is updated once per second.</td>
</tr>
<tr>
<td>DitherTest</td>
<td>Demonstrates drawing with a graphics technique known as dithering that</td>
</tr>
<tr>
<td></td>
<td>allows gradual transformation from one color to another.</td>
</tr>
</tbody>
</table>

Fig. 3.1 The examples from the applets directory (part 1 of 2).
Change directories to subdirectory \texttt{TicTacToe}. In that directory you will find the HTML file \texttt{example1.html} that is used to execute the applet. In the command window, type the command

\texttt{appletviewer example1.html}

and press the \textit{Enter} key. This executes the \texttt{appletviewer}. The \texttt{appletviewer} loads the HTML file specified as its \textit{command-line argument} (\texttt{example1.html}), determines from the file which applet to load (we discuss the details of HTML files in Section 3.3) and begins executing the applet. Figure 3.2 shows several screen captures of playing Tic-Tac-Toe with this applet.
Testing and Debugging Tip 3.3

If the `appletviewer` command does not work and/or the system indicates that the `appletviewer` command cannot be found, the `PATH` environment variable may not be defined properly on your computer. Review the installation directions for the Java 2 Software Development Kit to ensure that the `PATH` environment variable is defined correctly for your system (on some computers, you may need to restart your computer after modifying the `PATH` environment variable).

You are player X. To interact with the applet, point the mouse at the square where you want to place an X and click the mouse button (normally, the left mouse button). The applet plays a sound (assuming your computer supports audio playback) and places an X in the square if the square is open. If the square is occupied, this is an invalid move and the applet plays a different sound indicating that you cannot make the specified move. After you make a valid move, the applet responds by making its own move (this happens quickly).

To play again, re-execute the applet by clicking the `appletviewer`'s Applet menu and selecting the `Reload` menu item (Fig. 3.3). To terminate the `appletviewer`, click the `appletviewer`'s Applet menu and select the `Quit` menu item.
3.2.2 The DrawTest Applet

The next applet we demonstrate allows you to draw lines and points in different colors. To draw, you simply drag the mouse on the applet by pressing a mouse button and holding it while you drag the mouse. For this example, change directories to directory applets, then to subdirectory DrawTest. In that directory is the example1.html file that is used to execute the applet. In the command window, type the command

```
appletviewer example1.html
```

and press the Enter key. This executes the appletviewer. The appletviewer loads the HTML file specified as its command-line argument (example1.html again), determines from the file which applet to load and begins execution of the applet. Figure 3.4 shows a screen capture of this applet after drawing some lines and points.

The default shape to draw is a line and the default color is black, so you can draw black lines by dragging the mouse across the applet. To drag the mouse, press and hold the mouse button and move the mouse. Notice that the line follows the mouse pointer around the applet. The line is not permanent until you release the mouse button. You can then start a new line by repeating the process.

Select a color by clicking the circle inside one of the colored rectangles at the bottom of the applet. You can select from red, green, blue, pink, orange and black. The GUI components used to present these options are commonly known as radio buttons. If you think of a car radio, only one radio station can be selected at a time. Similarly, only one drawing color can be selected at a time.

![Sample execution of applet DrawTest.](image-url)
Try changing the shape to draw from **Lines** to **Points** by clicking the down arrow to the right of the word **Lines** at the bottom of the applet. A list drops down from the GUI component containing the two choices—**Lines** and **Points**. To select **Points**, click the word **Points** in the list. The GUI component closes the list and the current shape type is now **Points**. This GUI component is commonly known as a *choice, combo box or dropdown list*.

To start a new drawing, select **Reload** from the appletviewer’s **Applet** menu. To terminate the applet, select **Quit** from the appletviewer’s **Applet** menu.

### 3.2.3 The Java2D Applet

The last applet we demonstrate before defining applets of our own shows many of the complex new two-dimensional drawing capabilities built into Java 2—known as the **Java2D API**. For this example, change directories to the **jfc** directory in the J2SDK’s **demo** directory, then change to the **Java2D** directory (you can move up the directory tree toward **demo** using the command “`cd ..`” in both Windows and UNIX/Linux). In that directory is an HTML file (**Java2Demo.html**) that is used to execute the applet. In the command window, type the command

```
appletviewer Java2Demo.html
```

and press the Enter key. This executes the **appletviewer**. The **appletviewer** loads the HTML file specified as its command-line argument (**Java2Demo.html**), determines from the file which applet to load and begins execution of the applet. This particular demo takes some time to load as it is quite large. Figure 3.5 shows a screen capture of one of this applet’s many demonstrations of Java’s two-dimensional graphics capabilities.

At the top of this demo you see tabs that look like file folders in a filing cabinet. This demo provides 12 different tabs with several different features on each tab. To change to a different part of the demo, simply click one of the tabs. Also, try changing the options in the upper-right corner of the applet. Some of these affect the speed with which the applet draws the graphics. For example, click the small box with a check in it (a GUI component known as a **checkbox**) to the left of the word **Anti-Aliasing** to turn off anti-aliasing (a graphics technique for producing smoother on-screen graphics in which the edges of the graphic are blurred). When this feature is turned off (i.e., its **checkbox** is unchecked), the animation speed increases for the animated shapes at the bottom of the demo shown in Fig. 3.5. This occurs because an animated shape displayed with anti-aliasing takes longer to draw than an animated shape without anti-aliasing.

### 3.3 A Simple Java Applet: Drawing a String

Now, let’s get started with some applets of our own. Remember, we are just getting started—we have many more topics to learn before we can write applets similar to those demonstrated in Section 3.2. However, we will cover many of the same techniques in this book.

We begin by considering a simple applet that mimics the program of Fig. 2.1 by displaying the string "**Welcome to Java Programming!**". The applet and its screen output are shown in Fig. 3.6. The HTML document to load the applet into the **appletviewer** is shown and discussed in Fig. 3.7.
Fig. 3.5 Sample execution of applet **Java2D**.

```java
// Fig. 3.6: WelcomeApplet.java
// A first applet in Java.

// Java core packages
import java.awt.Graphics;    // import class Graphics

// Java extension packages
import javax.swing.JApplet;  // import class JApplet

public class WelcomeApplet extends JApplet {
    public void paint(Graphics g) {
        // call inherited version of method paint
        super.paint(g);
    }
}
```

Fig. 3.6 A first applet in Java and the applet’s screen output (part 1 of 2).
This program illustrates several important Java features. We consider each line of the program in detail. Line 19 does the “real work” of the program, namely drawing the string Welcome to Java Programming! on the screen. But let us consider each line of the program in order. Lines 1–2

// Fig. 3.6: WelcomeApplet.java
// A first applet in Java.

begin with //, indicating that the remainder of each line is a comment. The comment on line 1 indicates the figure number and file name for the applet source code. The comment on line 2 simply describes the purpose of the program.

As stated in Chapter 2, Java contains many predefined components called classes that are grouped into packages in the Java API. Line 5

import java.awt.Graphics;    // import class Graphics

is an import statement that tells the compiler load class Graphics from package java.awt for use in this Java applet. Class Graphics enables a Java applet to draw graphics such as lines, rectangles, ovals and strings of characters. Later in the book, you will see that class Graphics also enables applications to draw.

Line 8

import javax.swing.JApplet;  // import class JApplet

is an import statement that tells the compiler load class JApplet from package javax.swing. When you create an applet in Java, you normally import the JApplet class. [Note: There is an older class called Applet from package java.applet that is not used with Java’s newest GUI components from the javax.swing package. In this book, we use only class JApplet with applets.]
As with applications, every Java applet you create contains at least one class definition. One key feature of class definitions that was not mentioned in Chapter 2 is that programmers rarely create class definitions “from scratch.” In fact, when you create a class definition, you normally use pieces of an existing class definition. Java uses inheritance (introduced in Section 1.15 and discussed in detail in Chapter 9, “Object-Oriented Programming”) to create new classes from existing class definitions. Line 10

```java
public class WelcomeApplet extends JApplet {
```

begins a class definition for class WelcomeApplet. At the end of line 10, the left brace, {, begins the body of the class definition. The corresponding right brace, }, on line 23 ends the class definition. Keyword class introduces the class definition. WelcomeApplet is the class name. Keyword extends indicates that class WelcomeApplet inherits existing pieces from another class. The class from which WelcomeApplet inherits (JApplet) appears to the right of extends. In this inheritance relationship, JApplet is called the superclass or base class and WelcomeApplet is called the subclass or derived class. Using inheritance here results in a WelcomeApplet class definition that has the attributes (data) and behaviors (methods) of class JApplet as well as the new features we are adding in our WelcomeApplet class definition (specifically, the ability to draw Welcome to Java Programming! on the applet).

A key benefit of extending class JApplet is that someone else previously defined “what it means to be an applet.” The appletviewer and World Wide Web browsers that support applets expect every Java applet to have certain capabilities (attributes and behaviors). Class JApplet already provides all those capabilities—programmers do not need to “reinvent the wheel” and define all these capabilities on their own. In fact, applet containers expect applets to have over 200 different methods. In our programs to this point, we defined one method in each program. If we had to define over 200 methods just to display Welcome to Java Programming!, we would never create an applet, because it would take too long to define one! Using extends to inherit from class JApplet enables applet programmers to create new applets quickly.

The inheritance mechanism is easy to use; the programmer does not need to know every detail of class JApplet or any other superclass from which a new class inherits. The programmer needs to know only that class JApplet defines the capabilities required to create the minimum applet. However, to make the best use of any class, programmers should study all the capabilities of the superclass.

**Good Programming Practice 3.1**

Investigate the capabilities of any class in the Java API documentation ([java.sun.com/j2se/1.3/docs/api/index.html](http://java.sun.com/j2se/1.3/docs/api/index.html)) carefully before inheriting a subclass from it. This helps ensure that the programmer does not unintentionally “reinvent the wheel” by redefining a capability that the superclass already provides.

Classes are used as “templates” or “blueprints” to instantiate (or create) objects for use in a program. An object (or instance) resides in the computer’s memory and contains information used by the program. The term object normally implies that attributes (data) and behaviors (methods) are associated with the object. The object’s methods use the attributes to provide useful services to the client of the object (i.e., the code in a program that calls the methods).
When an applet container (the appletviewer or browser in which the applet executes) loads our WelcomeApplet class, the applet container creates an object (instance) of class WelcomeApplet that implements the applet’s attributes and behaviors. [Note: The terms instance and object are often used interchangeably.] Applet containers can create only objects of classes that are public and extend JApplet. Thus, applet containers require applet class definitions to begin with the keyword public (line 10). Otherwise, the applet container cannot load and execute the applet. The public keyword and related keywords (such as private and protected) are discussed in detail in Chapter 8, “Object-Based Programming.” For now, we ask you simply to start all class definitions with the public keyword until the discussion of public in Chapter 8.

When you save a public class in a file, the file name must be the class name followed by the .java file name extension. For our applet, the file name must be WelcomeApplet.java. Please note that the class name part of the file name must use the same spelling as the class name, including identical use of uppercase and lowercase letters. For reinforcement, we repeat two Common Programming Errors from Chapter 2.

**Common Programming Error 3.1**

It is an error for a public class if the file name is not identical to the class name (plus the .java extension) in both spelling and capitalization. Therefore, it is also an error for a file to contain two or more public classes.

**Common Programming Error 3.2**

It is an error not to end a file name with the .java extension for a file containing an application’s class definition. The Java compiler will not be able to compile the class definition.

**Testing and Debugging Tip 3.4**

The compiler error message “Public class ClassName must be defined in a file called ClassName.java” indicates either that the file name does not exactly match the name of the public class in the file (including all uppercase and lowercase letters) or that you typed the class name incorrectly when compiling the class (the name must be spelled with the proper uppercase and lowercase letters).

Line 13

```java
public void paint( Graphics g )
```

begins the definition of the applet’s paint method—one of three methods (behaviors) that the applet container calls for an applet when the container begins executing the applet. In order, these three methods are init (discussed later in this chapter), start (discussed in Chapter 6) and paint. Your applet class gets a “free” version of each of these methods from class JApplet when you specify extends JApplet in the first line of your applet’s class definition. If you do not define these methods in your own applet, the applet container calls the versions inherited from JApplet. The inherited versions of methods init and start have empty bodies (i.e., their bodies do not contain statements, so they do not perform a task) and the inherited version of method paint does not draw anything on the applet. [Note: There are several other methods that an applet container calls during an applet’s execution. We discuss these methods in Chapter 6, “Methods.”]

To enable our applet to draw, class WelcomeApplet overrides (replaces or redefines) the default version of paint by placing statements in the body of paint that draw
a message on the screen. When the applet container tells the applet to “draw itself” on the screen by calling method `paint`, our message `Welcome to Java Programming!` appears rather than a blank screen.

Lines 13–21 are the definition of `paint`. The task of method `paint` is to draw graphics (such as lines, ovals and strings of characters) on the screen. Keyword `void` indicates that this method does not return any results when it completes its task. The set of parentheses after `paint` defines the method’s parameter list. The parameter list is where methods receive data required to perform their tasks. Normally, this data is passed by the programmer to the method through a method call (also known as invoking a method or sending a message). For example, in Chapter 2 we passed data to `JOptionPane`’s `showMessageDialog` method such as the message to display or the type of dialog box. However, when writing applets, the programmer does not call method `paint` explicitly. Rather, the applet container calls `paint` to tell the applet to draw and the applet container passes to the `paint` method the information `paint` requires to perform its task, namely a `Graphics` object (called `g`). It is the applet container’s responsibility to create the `Graphics` object to which `g` refers. Method `paint` uses the `Graphics` object to draw graphics on the applet. The public keyword at the beginning of line 13 is required so the applet container can call your `paint` method. For now, all method definitions should begin with the public keyword. We introduce other alternatives in Chapter 8.

The left brace, `{`, on line 14 begins method `paint`’s body. The corresponding right brace, `}`, on line 21 ends `paint`’s body.

Line 16

```java
super.paint( g );
```
calls the version of method `paint` inherited from superclass `JApplet`.¹

Line 19

```java
g.drawString( "Welcome to Java Programming!", 25, 25 );
```
instructs the computer to perform an action (or task), namely to draw the characters of the string `Welcome to Java Programming!` on the applet. This statement uses method `drawString` defined by class `Graphics` (this class defines all the drawing capabilities of a Java program, including strings of characters and shapes such as rectangles, ovals and lines). The statement calls method `drawString` using the `Graphics` object `g` (in `paint`’s parameter list) followed by a dot operator (.) followed by the method name `drawString`. The method name is followed by a set of parentheses containing the argument list `drawString` needs to perform its task.

The first argument to `drawString` is the `String` to draw on the applet. The last two arguments in the list—25 and 25—are the x-y coordinates (or position) at which the bottom-left corner of the string should be drawn on the applet. Drawing methods from class `Graphics` require coordinates to specify where to draw on the applet (later in the text we

---

¹. For reasons that will become clear later in the text, this statement should be the first statement in every applet’s `paint` method. Although the early examples of applets will work without this statement, omitting this statement causes subtle errors in more elaborate applets that combine drawing and GUI components. Including this statement now will get you in the habit of using it and will save time and effort as you build more substantial applets later.
demonstrate drawing in applications). The first coordinate is the x-coordinate (the number of pixels from the left side of the applet), and the second coordinate is the y-coordinate (representing the number of pixels from the top of the applet). Coordinates are measured from the upper-left corner of the applet in pixels (just below the Applet menu in the sample output window of Fig. 3.6). A pixel (“picture element”) is the unit of display for your computer’s screen. On a color screen, a pixel appears as one colored dot on the screen. Many personal computers have 800 pixels for the width of the screen and 600 pixels for the height of the screen, for a total of 800 times 600 or 480,000 displayable pixels. Many computers today have screens with higher screen resolutions, i.e., they have more pixels for the width and height of the screen. The size of an applet on the screen depends on the size and resolution of the screen. For screens with the same size, the applet will appear smaller on the screen with the higher resolution. Note that some older computers have screen resolutions lower than 800 by 600. The most common lower resolution is 640 by 480.

When line 19 executes, it draws the message Welcome to Java Programming! on the applet at the coordinates 25 and 25. Note that the quotation marks enclosing the string are not displayed on the screen.

As an aside, why would you want free copies of methods init, start and paint if they do not perform a task? The predefined start-up sequence of method calls made by the appletviewer or browser for every applet is always init, start and paint—this provides an applet programmer a guarantee that these methods will be called as every applet begins execution. Every applet does not need all three of these methods. However, the appletviewer or browser expects each of these methods to be defined so it can provide a consistent start-up sequence for an applet. [Note: This is similar to applications always starting execution with main.] Inheriting the default versions of these methods guarantees the browser that it can treat each applet uniformly by calling init, start and paint as applet execution begins. Also, the programmer can concentrate on defining only the methods required for a particular applet.

### 3.3.1 Compiling and Executing WelcomeApplet

As with application classes, you must compile applet classes before they can execute. After defining class WelcomeApplet and saving it in WelcomeApplet.java, open a command window, change to the directory in which you saved the applet class definition and type the command

```
javac WelcomeApplet.java
```

to compile class WelcomeApplet. If there are no syntax errors, the resulting bytecodes are stored in the file WelcomeApplet.class.

Before you can execute the applet you must create an HTML (Hypertext Markup Language) file to load the applet into the applet container (appletviewer or a browser). Typically, an HTML file ends with the “.html” or “.htm” file name extension. Browsers display the contents of documents that contain text (also known as text files). To execute a Java applet, an HTML text file must indicate which applet the applet container should load and execute. Figure 3.7 shows a simple HTML file—WelcomeApplet.html—that loads into the applet defined in Fig. 3.6 into an applet container. [Note: For the early part of this book, we always demonstrate applets with the appletviewer applet container.]
Good Programming Practice 3.2

Always test a Java applet in the appletviewer and ensure that it is executing correctly before loading the applet into a World Wide Web browser. Browsers often save a copy of an applet in memory until the current browsing session terminates (i.e., all browser windows are closed). Thus, if you change an applet, recompile the applet, then reload the applet in the browser, you may not see the changes because the browser may still be executing the original version of the applet. Close all your browser windows to remove the old version of the applet from memory. Open a new browser window and load the applet to see your changes.

Software Engineering Observation 3.1

If your World Wide Web browser does not support Java 2, most of the applets in this book will not execute in your browser. This is because most of the applets in this book use features that are specific to Java 2 or are not provided by browsers that support Java 1.1. Section 3.6.2 discusses how to use the Java Plug-in to view applets in Web browsers that do not support Java 2.

Many HTML codes (or tags) come in pairs. For example, lines 1 and 4 of Fig. 3.7 indicate the beginning and the end, respectively, of the HTML tags in the file. All HTML tags begin with a left angle bracket, <, and end with a right angle bracket, >. Lines 2 and 3 are special HTML tags for Java applets. They tell the applet container to load a specific applet and define the size of the applet’s display area (its width and height in pixels) in the appletviewer (or browser). Normally, the applet and its corresponding HTML file are stored in the same directory on disk. Typically, a browser loads an HTML file from a computer (other than your own) connected to the Internet. However, HTML files also can reside on your computer (as we demonstrated in Section 3.2). When an applet container encounters an HTML file that specifies an applet to execute, the applet container automatically loads the applet’s .class file (or files) from the same directory on the computer in which the HTML file resides.

The <applet> tag has several attributes. The first attribute of the <applet> tag on line 2 (code = "WelcomeApplet.class") indicates that the file WelcomeApplet.class contains the compiled applet class. Remember, when you compile your Java programs, every class is compiled into a separate file that has the same name as the class and ends with the .class extension. The second and third attributes of the <applet> tag indicate the width and the height of the applet in pixels. The upper-left corner of the applet’s display area is always at x-coordinate 0 and y-coordinate 0. The width of this applet is 300 pixels and its height is 45 pixels. You may want (or need) to use larger width and height values to define a larger drawing area for your applets. The </applet> tag (line 3) terminates the <applet> tag that began on line 2. The </html> tag (line 4) specifies the end of the HTML tags that began on line 1 with <html>.
Software Engineering Observation 3.2

Generally, each applet should be less than 800 pixels wide and 600 pixels tall (most computer screens support these dimensions as the minimum width and height).

Common Programming Error 3.3

Placing additional characters such as commas (,) between the attributes in the `<applet>` tag may cause the `appletviewer` or browser to produce an error message indicating a `MissingResourceException` when loading the applet.

Common Programming Error 3.4

Forgetting the ending `<applet>` tag prevents the applet from loading into the `appletviewer` or browser properly.

Testing and Debugging Tip 3.5

If you receive a `MissingResourceException` error message when loading an applet into the `appletviewer` or a browser, check the `<applet>` tag in the HTML file carefully for syntax errors. Compare your HTML file to the file in Fig. 3.7 to confirm proper syntax.

The `appletviewer` understands only the `<applet>` and `</applet>` HTML tags, so it is sometimes referred to as the “minimal browser” (it ignores all other HTML tags). The `appletviewer` is an ideal place to test an applet and ensure that it executes properly. Once the applet’s execution is verified, you can add the applet’s HTML tags to an HTML file that will be viewed by people browsing the Internet.

To execute the `WelcomeApplet` in the `appletviewer` open a command window, change to the directory containing your applet and HTML file and type the command

```
appletviewer WelcomeApplet.html
```

Note that the `appletviewer` requires an HTML file to load an applet. This is different from the `java` interpreter for applications which requires only the class name of the application class. Also, the preceding command must be issued from the directory in which the HTML file and the applet’s `.class` file are located.

Common Programming Error 3.5

Running the `appletviewer` with a file name that does not end with `.html` or `.htm` is an error that prevents the `appletviewer` from loading your applet for execution.

Portability Tip 3.2

Test your applets in every browser used by people who view your applet. This will help ensure that people who view your applet experience the functionality you expect. [Note: A goal of the Java Plug-In (discussed later in the book) is to provide consistent applet execution across many different browsers.]

3.4 Two More Simple Applets: Drawing Strings and Lines

Let us consider another applet. An applet can draw `Welcome to Java Programming`! several ways. For example, an applet can use two `drawString` statements in method `paint` to print multiple lines of text as in Fig. 3.8. The HTML file to load the applet into an applet container is shown in Fig. 3.9.
Note that each call to method `drawString` can draw at any pixel location on the applet. The reason the two output lines appear left aligned as shown in Fig. 3.8 is that both use the same x coordinate (25). Also, each `drawString` method call uses different y coordinates (25 on line 19 and 40 on line 20), so the strings appear at different vertical locations on the applet. If we reverse lines 19 and 20 in the program, the output window will still appear as shown because the pixel coordinates specified in each `drawString` statement are independent of the coordinates specified in all other `drawString` statements (and all other drawing operations). When drawing graphics, lines of text are not separated by newline characters (as shown with methods `System.out`’s method `println` and `JOptionPane`’s method `showMessageDialog` in Chapter 2). In fact, if you try
to output a string containing a newline character (\n), you will simply see a small black box at that position in the string.

To make drawing more interesting, the applet of Fig. 3.10 draws two lines and a string. The HTML file to load the applet into an applet container is shown in Fig. 3.11.

```java
1 // Fig. 3.10: WelcomeLines.java
2 // Displaying text and lines
3
4 // Java core packages
5 import java.awt.Graphics;    // import class Graphics
6
7 // Java extension packages
8 import javax.swing.JApplet;  // import class JApplet
9
10 public class WelcomeLines extends JApplet {
11     // draw lines and a string on applet’s background
12     public void paint( Graphics g )
13     {
14         // call inherited version of method paint
15         super.paint( g );
16
17         // draw horizontal line from (15, 10) to (210, 10)
18         g.drawLine( 15, 10, 210, 10 );
19
20         // draw horizontal line from (15, 30) to (210, 30)
21         g.drawLine( 15, 30, 210, 30 );
22
23         // draw String between lines at location (25, 25)
24         g.drawString( "Welcome to Java Programming!", 25, 25 );
25
26     } // end method paint
27
28 } // end class WelcomeLines
```

Fig. 3.10  Drawing strings and lines.

```html
1 <html>
2 <applet code = "WelcomeLines.class" width = "300" height = "40">
3 </applet>
4 </html>
```

Fig. 3.11  The WelcomeLines.html file, which loads class WelcomeLines of Fig. 3.10 into the appletviewer.
Lines 19 and 22 of method \texttt{paint}

\begin{verbatim}
g.drawLine( 15, 10, 210, 10 );
g.drawLine( 15, 30, 210, 30 );
\end{verbatim}

use \textit{method} \texttt{drawLine} of class \texttt{Graphics} to indicate that the \texttt{Graphics} object that \texttt{g} refers to should draw lines. Method \texttt{drawLine} requires four arguments that represent the two end points of the line on the applet—the \( x \)-coordinate and \( y \)-coordinate of the first end point in the line and the \( x \)-coordinate and \( y \)-coordinate of the second end point in the line. All coordinate values are specified with respect to the upper-left corner \((0, 0)\) coordinate of the applet. Method \texttt{drawLine} draws a straight line between the two end points.

\section*{3.5 Another Java Applet: Adding Floating-Point Numbers}

Our next applet (Fig. 3.12) mimics the application of Fig. 2.9 for adding two integers. However, this applet requests that the user enter two \textit{floating-point numbers} (i.e., numbers with a decimal point such as 7.33, 0.0975 and 1000.12345). To store floating-point numbers in memory we introduce primitive data type \texttt{double}, which represents \textit{double-precision floating-point} numbers. There is also primitive data type \texttt{float} for \textit{storing single-precision floating-point} numbers. A \texttt{double} requires more memory to store a floating-point value, but stores it with approximately twice the precision of a \texttt{float} (15 significant digits for \texttt{double} vs. seven significant digits for \texttt{float}).

Once again, we use \texttt{JOptionPane.showInputDialog} to request input from the user. The applet computes the sum of the input values and displays the result by drawing a string inside a rectangle on the applet. The HTML file to load this applet into the \texttt{applet-viewer} is shown in Fig. 3.13.

```java
// Fig. 3.12: AdditionApplet.java
// Adding two floating-point numbers.

// Java core packages
import java.awt.Graphics;   // import class Graphics

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

public class AdditionApplet extends JApplet {

   double sum;  // sum of values entered by user

   // initialize applet by obtaining values from user
   public void init()
   {
      String firstNumber;   // first string entered by user
      String secondNumber;  // second string entered by user
      double number1;       // first number to add
      double number2;       // second number to add

      // obtain first number from user
      firstNumber = JOptionPane.showInputDialog(
         "Enter first floating-point value" );
```
// obtain second number from user
secondNumber = JOptionPane.showInputDialog(
    "Enter second floating-point value" );

// convert numbers from type String to type double
number1 = Double.parseDouble( firstNumber );
number2 = Double.parseDouble( secondNumber );

// add numbers
sum = number1 + number2;

// draw results in a rectangle on applet’s background
public void paint( Graphics g )
{
    // call inherited version of method paint
    super.paint( g );

    // draw rectangle starting from (15, 10) that is 270
    // pixels wide and 20 pixels tall
    g.drawRect( 15, 10, 270, 20 );

    // draw results as a String at (25, 25)
    g.drawString( "The sum is " + sum, 25, 25 );
}

}  // end method paint

} // end class AdditionApplet

Fig. 3.12 An addition program “in action” (part 2 of 2).
// Fig. 3.12: AdditionApplet.java  
// Adding two floating-point numbers.

// import class Graphics  
import java.awt.Graphics;  
// import class Graphics

// import package javax.swing  
import javax.swing.*;  
// import package javax.swing

// import package javax.swing  
import javax.swing.*;  
// import package javax.swing

Software Engineering Observation 3.3  
The Java compiler does not require import statements in a Java source code file if the complete class name—the full package name and class name (e.g., java.awt.Graphics)—is specified every time a class name is used in the source code.

Software Engineering Observation 3.4  
The compiler does not load every class in a package when it encounters an import statement that uses the * (e.g., javax.swing.*) notation. The compiler loads from the package only those classes the program uses.
Software Engineering Observation 3.5

Many packages have subpackages. For example, the `java.awt` package has subpackage `event` for the package `java.awt.event`. When the compiler encounters an `import` statement that uses the `*` (e.g., `java.awt.*`) notation to indicate that a program uses multiple classes from a package, the compiler does not load classes from the subpackage `event`. Thus, you cannot define an `import` of `java.*` to search for classes from all Java core packages.

Software Engineering Observation 3.6

When using `import` statements, separate `import` statements must be specified for each package used in a program.

Common Programming Error 3.6

Assuming that an `import` statement for an entire package (e.g., `java.awt.*`) also `import`s classes from subpackages in that package (e.g., `java.awt.event.*`) results in syntax errors for the classes from the subpackages. There must be separate import statements for every package from which classes are used.

Remember that applets inherit from the `JApplet` class, so they have all the methods that an applet container requires to execute the applet. Line 10

```java
public class AdditionApplet extends JApplet {
```

begins class `AdditionApplet`'s definition and indicates that it inherits from `JApplet`.

All class definitions start with an opening left brace (end of line 10), `{`, and end with a closing right brace, `}` (line 52).

Common Programming Error 3.7

If braces do not occur in matching pairs, the compiler indicates a syntax error.

Line 11

```java
double sum;  // sum of values entered by user
```

is an instance variable declaration—every instance (object) of the class contains one copy of each instance variable. For example, if there are 10 instances of this applet executing, each instance has its own copy of `sum`. Thus, there would be 10 separate copies of `sum` (one per applet). Programmers declare instance variables in the body of the class definition, but outside the bodies of all the class’s method definitions. The preceding declaration states that `sum` is a variable of primitive type `double`.

A benefit of instance variables is that all the methods of the class can use the instance variables. Until now, we declared all variables in an application’s `main` method. Variables defined in the body of a method are known as `local variables` and can be used only in the body of the method in which they are defined. Another distinction between instance variables and local variables is that instance variables have default values and local variables do not. The default value of variable `sum` is 0.0, because `sum` is an instance variable.

Good Programming Practice 3.3

Explicitly initializing instance variables rather than relying on automatic initialization improves program readability.
The applet of Fig. 3.12 contains two methods—\texttt{init} (lines 14–35) and \texttt{paint} (lines 38–50). When an applet container loads an applet, the container creates an instance of the applet class and calls its \texttt{init} method. The applet container calls method \texttt{init} only once during an applet’s execution. Method \texttt{init} normally initializes the applet’s instance variables (if they need to be initialized to a value other than their default value) and performs tasks that should occur only once when the applet begins execution. As we will see in later chapters, the applet’s \texttt{init} method typically creates the applet’s graphical user interface.

Software Engineering Observation 3.7

The order in which methods are defined in a class definition has no effect on when those methods are called at execution time. However, following conventions for the order in which methods are defined improves program readability and maintainability.

The first line of the \texttt{init} method always appears as

\begin{verbatim}
public void init()
\end{verbatim}

indicating that \texttt{init} is a public method that returns no information (\texttt{void}) when it completes and receives no arguments (empty parentheses after \texttt{init}) to perform its task.

The left brace (line 15) marks the beginning of \texttt{init}’s body, and the corresponding right brace (line 35) marks the end of \texttt{init}. Lines 16–17

\begin{verbatim}
String firstNumber;   // first string entered by user
String secondNumber;  // second string entered by user
\end{verbatim}

declare local \texttt{String} variables \texttt{firstNumber} and \texttt{secondNumber} in which the program stores the \texttt{Strings} input by the user.

Lines 18–19

\begin{verbatim}
double number1;       // first number to add
double number2;       // second number to add
\end{verbatim}

declare local variables \texttt{number1} and \texttt{number2} of primitive data type \texttt{double}—these variables hold floating-point values. Unlike \texttt{sum, number1} and \texttt{number2} are not instance variables, so they are not initialized to 0.0 (the default value of \texttt{double} instance variables).

As an important aside, there are actually two types of variables in Java—\textit{primitive data type variables} (normally called \textit{variables}) and \textit{reference variables} (normally called \textit{references}). The identifiers \texttt{firstNumber} and \texttt{secondNumber} are actually references—names that are used to refer to objects in the program. Such references actually contain the location of an object in the computer’s memory. In our preceding applets, method \texttt{paint} actually receives a reference called \texttt{g} that refers to a \texttt{Graphics} object. Statements in method \texttt{paint} use that reference to send messages to the \texttt{Graphics} object. These messages are calls to methods (like \texttt{drawString}, \texttt{drawLine} and \texttt{drawRect}) that enable the program to draw. For example, the statement

\begin{verbatim}
g.drawString( "Welcome to Java Programming!", 25, 25 );
\end{verbatim}

sends the \texttt{drawString} message to the \texttt{Graphics} object to which \texttt{g} refers. As part of the message, which is simply a method call, we provide the data that \texttt{drawString} re-

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
quires to do its task. The Graphics object uses this data to draw the String at the specified location.

The identifiers number1, number2 and sum are the names of variables. A variable is similar to an object. The primary difference between a variable and an object is that an object is defined by a class definition that can contain both data (instance variables) and methods, whereas a variable is defined by a primitive (or built-in) data type (one of char, byte, short, int, long, float, double or boolean) that can contain only data. A variable can store exactly one value at a time, whereas one object may contain many individual pieces of data. The distinction between a variable and a reference is based on the data type of the identifier, which is stated in a declaration. If the data type is a class name, the identifier is a reference to an object and that reference can be used to send messages to (call methods on) that object. If the data type is one of the primitive data types, the identifier is a variable that can be used to store in memory or retrieve from memory a single value of the declared primitive type.

Software Engineering Observation 3.8

A hint to help you determine if an identifier is a variable or a reference is the variable’s data type. By convention all class names in Java start with a capital letter. Therefore, if the data type starts with a capital letter, normally you can assume that the identifier is a reference to an object of the declared type (e.g., Graphics g indicates that g is a reference to a Graphics object).

Lines 22–23

// obtain first number from user
firstNumber = JOptionPane.showInputDialog(
    "Enter first floating-point value" );

read the first floating-point number from the user. JOptionPane method showInputDialog displays an input dialog that prompts the user to enter a value. The user types a value in the input dialog’s text field, then clicks the OK button to return the string the user typed to the program. If you type and nothing appears in the text field, position the mouse pointer in the text field and click the mouse to make the text field active. Variable firstNumber is assigned the result of the call to JOptionPane.showInputDialog operation with an assignment statement. The statement is read as “firstNumber gets the value of JOptionPane.showInputDialog( "Enter first floating-point value" ).”

In lines 22–23, notice the method call syntax. At this point, we have seen two different ways to call methods. This statement uses the static method call syntax introduced in Chapter 2. All static methods are called with the syntax

ClassName.methodName( arguments )

Also in this chapter, we have called methods of class Graphics with a similar syntax that started with a reference to a Graphics object. Generically, this syntax is

referenceName.methodName( arguments )

This syntax is used for most methods calls in Java. In fact, the applet container uses this syntax to call methods init, start and paint on your applets.
Lines 26–27

```java
// obtain second number from user
secondNumber = JOptionPane.showInputDialog(
    "Enter second floating-point value" );
```

read the second floating-point value from the user by displaying an input dialog.

Lines 30–31

```java
number1 = Double.parseDouble( firstNumber );
number2 = Double.parseDouble( secondNumber );
```

convert the two strings input by the user to `double` values for use in a calculation. Method `Double.parseDouble` (a `static` method of class `Double`) converts its `String` argument to a `double` floating-point value. Class `Double` is in package `java.lang`. The floating-point value returned by `parseFloatDouble` in line 30 is assigned to variable `number1`. The floating-point value returned by `parseFloatDouble` in line 31 is assigned to variable `number2`.

Software Engineering Observation 3.9

> Each primitive data type (such as `int` or `double`) has a corresponding class (such as `Integer` or `Double`) in package `java.lang`. These classes (commonly known as type-wrappers) provide methods for processing primitive data type values (such as converting a `String` to a primitive data type value or converting a primitive data type value to a `String`). Primitive data types do not have methods. Therefore, methods related to a primitive data type are located in the corresponding type-wrappers (e.g., method `parseFloatDouble` that converts a `String` to a `double` value is located in class `Double`). See the online API documentation for the complete details of the methods in the type-wrappers classes.

The assignment statement at line 34

```java
sum = number1 + number2;
```

calculates the sum of the values stored in variables `number1` and `number2` and assigns the result to variable `sum` using the assignment operator `=`. The statement is read as “`sum` gets the value of `number1 + number2`.” Notice that instance variable `sum` is used in method `init` even though `sum` was not defined in method `init`. We can use `sum` in `init` (and all other methods of the class), because `sum` is an instance variable.

At this point the applet’s `init` method returns and the applet container calls the applet’s `start` method. We did not define `start` in this applet, so the one inherited from class `JApplet` is called here. Normally, the `start` method is used with an advanced concept called multithreading. See Chapter 15 and Chapter 18 for typical uses of `start`.

Next, the applet container calls the applet’s `paint` method. In this example, method `paint` draws a rectangle in which the result of the addition will appear. Line 45

```java
g.drawRect( 15, 10, 270, 20 );
```

sends the `drawRect` message to the `Graphics` object to which `g` refers (calls the `Graphics` object’s `drawRect` method). Method `drawRect` draws a rectangle based on its four arguments. The first two integer values represent the `upper-left x-coordinate` and `upper-left y-coordinate` where the `Graphics` object begins drawing the rectangle. The third and fourth arguments are non-negative integers that represent the `width` of the rectan-
gle in pixels and the height of the rectangle in pixels, respectively. This particular statement draws a rectangle starting at coordinate (15, 10) that is 270 pixels wide and 20 pixels tall.

**Common Programming Error 3.8**

It is a logic error to supply a negative width or negative height as an argument to `Graphics` method `drawRect`. The rectangle will not be displayed and no error will be indicated.

**Common Programming Error 3.9**

It is a logic error to supply two points (i.e., pairs of x- and y-coordinates) as the arguments to `Graphics` method `drawRect`. The third argument must be the width in pixels and the fourth argument must be the height in pixels of the rectangle to draw.

**Common Programming Error 3.10**

It is normally a logic error to supply arguments to `Graphics` method `drawRect` that cause the rectangle to draw outside the applet’s viewable area (i.e., the width and height of the applet as specified in the HTML document that references the applet). Either increase the applet’s width and height in the HTML document or pass arguments to method `drawRect` that cause the rectangle to draw inside the applet’s viewable area.

Line 48

```java
    g.drawString( "The sum is " + sum, 25, 25 );
```

sends the `drawString` message to the `Graphics` object to which `g` refers (calls the `Graphics` object’s `drawString` method). The expression

```java
    "The sum is " + sum
```

from the preceding statement uses the string concatenation operator `+` to concatenate the string "The sum is " and `sum` (converted to a string) to create the string `drawString` displays. Notice again that the preceding statement uses the instance variable `sum` even though method `paint` does not define `sum` as a local variable.

The benefit of defining `sum` as an instance variable is that we were able to assign `sum` a value in `init` and use `sum`’s value in the `paint` method later in the program. All methods of a class are capable of using the instance variables in the class definition.

**Software Engineering Observation 3.10**

The only statements that should be placed in an applet’s `init` method are those that are directly related to the one-time initialization of an applet’s instance variables. The applet’s results should be displayed from other methods of the applet class. Results that involve drawing should be displayed from the applet’s `paint` method.

**Software Engineering Observation 3.11**

The only statements that should be placed in an applet’s `paint` method are those that are directly related to drawing (i.e., calls to methods of class `Graphics`) and the logic of drawing. Generally, dialog boxes should not be displayed from an applet’s `paint` method.

### 3.6 Viewing Applets in a Web Browser

We demonstrated several applets in this chapter using the `appletviewer` applet container. As we mentioned, applets also can execute in Java-enabled Web browsers. Unfortunate-
ly, there are many different browser versions being used worldwide. Some support only Java 1.0 and many support Java 1.1. However, few support the Java 2 Platform. Also, even the browsers that support Java 1.1 do so inconsistently. In Section 3.6.1, we demonstrate an applet executing in Netscape Navigator 6, which supports Java 2. In Section 3.6.2, we demonstrate how to use the Java Plug-in to execute Java 2 applets in other Web browsers such as Microsoft Internet Explorer or earlier versions of Netscape Navigator.

Portability Tip 3.3

Not all Web browsers support Java. Those that do often support different versions and are not always consistent across all platforms.

3.6.1 Viewing Applets in Netscape Navigator 6

When you install Netscape Navigator 6, one of the browser components in the default installation is Java 2. Once installed, you can simply load an applet’s HTML file into the browser to execute the applet. You can download and install Netscape 6 from

www.netscape.com

by clicking the Download button at the top of the Web page.

After installing the browser, open the program. On Windows, Netscape 6 typically places an icon on your desktop during the install process. In the File menu, click Open File... to select an HTML document from your local computer’s hard disk. In the Open File dialog, navigate to the location of the HTML file of Fig. 3.11. Select the file name WelcomeLines.html by clicking it, then click the Open button to open the file in the browser. In a few moments, you should see the applet of Fig. 3.10 appear in the browser window as shown in Fig. 3.14.

3.6.2 Viewing Applets in Other Browsers Using the Java Plug-In

If you would like to use the features of the Java 2 platform in an applet and execute that applet in a browser that does not support Java 2, Sun provides the Java Plug-in to bypass a browser’s Java support and use a complete version of the Java 2 Runtime Environment (J2RE) that is installed on the user’s local computer. If the J2RE does not already exist on the client machine, it can be downloaded and installed dynamically.

Performance Tip 3.1

Because of the size of the Java Plug-in, it is difficult and inefficient to download the Plug-in for users with slower Internet connections. For this reason, the Plug-in is ideal for corporate intranets where users are connected to a high-speed network. Once the Plug-in is downloaded, it does not need to be downloaded again.

You must indicate in the HTML file containing an applet that the browser should use the Java Plug-in to execute the applet. To do so, requires that you convert the <applet> and </applet> tags into tags that load the Java Plug-in and execute the applet. Sun provides a conversion utility called the Java Plug-in 1.3 HTML Converter that performs the conversion for you. Complete information on downloading and using the Java Plug-in and the HTML Converter are available at the Web site

java.sun.com/products/plugin/
Once you have downloaded and installed the Java Plug-in HTML converter, you can execute it via the batch file `HTMLConverter.bat` on Windows or the shell script `HTMLConverter.sh` on Linux/UNIX. These files are located in the converter directory’s `classes` subdirectory. Figure 3.15 shows the Java Plug-in HTML Converter window.

The Java Plug-in HTML Converter allows you to convert all the HTML files containing applets in one directory. Click the **Browse…** button to select the directory containing the files to convert. Also, you can specify the directory in which the original HTML files are saved.

2. As of Java 2 Software Development Kit version 1.3.1, a command-line version of the Java Plug-in HTML converter is one of the tools in the J2SDK. To use the command-line version, open a command window and change directories to the location that contains the HTML file to convert. In that directory type `HTMLConverter fileName`, where `fileName` is the HTML file to convert. Visit [java.sun.com/products/plugin/1.3/docs/htmlconv.html](http://java.sun.com/products/plugin/1.3/docs/htmlconv.html) for more details on the command-line HTML converter.
To perform the conversion, you must select the directory containing the HTML files to convert. You can either type the directory name in the text field below **All Files in Folder**, or you can select the directory by clicking the **Browse...** button to the right of that text field. We clicked the **Browse...** button to display the **Open** dialog in Fig. 3.16.

After selecting the directory containing files to convert, the **Java Plug in HTML Converter** window appears as in Fig. 3.17. The converter provides several conversion templates to support different combinations of browsers. The default template supports Netscape Navigator and Microsoft Internet Explorer. Figure 3.17 shows the expanded **Template File** drop-down list containing the pre-defined conversion templates. We selected the default template that enables Microsoft Internet Explorer and Netscape Navigator to use the plug-in to execute an applet.

After selecting the appropriate template file, click the **Convert...** button at the bottom of the **Java Plug in HTML Converter** window. Figure 3.18 shows the dialog box that appears containing the status and results of the conversion. At this point the applet’s HTML file can be loaded into Netscape Navigator or Microsoft Internet Explorer to execute the applet. If the Java 2 Runtime Environment does not already exist on the user’s computer, the converted HTML file contains information that enables the browser to prompt users to determine if they would like to download the plug-in.

In this chapter and Chapter 2, we have introduced many important features of Java, including applications, applets, displaying data on the screen, inputting data from the keyboard, performing calculations and making decisions. In Chapter 4, we build on these techniques as we introduce **structured programming**. Here, you will become more familiar with indentation techniques. We also study how to specify and vary the order in which a program executes statements—this order is called **flow of control**.

---

**Fig. 3.16** Selecting the directory containing HTML files to convert.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
If you have access to the Internet and the World Wide Web, there are a large number of Java applet resources available to you. The best place to start is at the source—the Sun Microsystems, Inc. Java Web site java.sun.com. In the upper-left corner of the Web page is an Applets hyperlink that takes you to the Web page java.sun.com/applets.

This page contains a variety of Java applet resources, including free applets you can use on your own World Wide Web site, the demonstration applets from the J2SDK and a variety of other applets (many of which can be downloaded and used on your own computer).
There is also a section entitled “Applets at Work” where you can read about uses of applets in industry.

On the Sun Microsystems Java Web site, visit the *Java Developer Connection*

```markdown
developer.java.sun.com/developer
```

This free site includes technical support, discussion forums, on-line training courses, technical articles, resources, announcements of new Java features, early access to new Java technologies, and links to other important Java Web sites. Even though the site is free, you must register to use it.

Another useful Web site is *JARS*—originally called the *Java Applet Rating Service*. The JARS site

```markdown
www.jars.com
```
calls itself the “#1 Java Review Service.” This site originally was a large Java repository for applets. Its benefit was that it rated every applet registered at the site as top 1%, top 5% and top 25%, so you could view the best applets on the Web. Early in the development of the Java language, having your applet rated here was a great way to demonstrate your Java programming abilities. JARS is now all-around resource for Java programmers.

The resources listed in this section provide hyperlinks to many other Java-related Web sites. If you have Internet access, spend some time browsing these sites, executing applets and reading the source code for the applets when it is available. This will help you rapidly expand your Java expertise. Appendix B contains many other Web-based Java resources.

### 3.8 (Optional Case Study) Thinking About Objects: Identifying the Classes in a Problem Statement

Now we begin the substantial task of designing the elevator simulator model, which represents the workings of the elevator system. We will design the user interaction and display of this model in Section 12.16 and Section 22.9, respectively.

**Identifying the Classes in a System**

The first step of our OOD process is to identify the classes in our model. We will eventually describe these classes in a formal way and implement them in Java. First, we review the problem statement and locate all the *nouns*; it is likely that these include most of the classes (or instances of classes) necessary to implement the elevator simulation. Figure 3.19 is a list of these nouns (and noun phrases) in the order of their appearance.

<table>
<thead>
<tr>
<th>Nouns (and noun phrases) in the problem statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>company</td>
</tr>
<tr>
<td>office building</td>
</tr>
<tr>
<td>elevator</td>
</tr>
<tr>
<td>software-simulator application</td>
</tr>
</tbody>
</table>

*Fig. 3.19* Nouns (and noun phrases) in problem statement (part 1 of 2).
We choose only the nouns that perform important duties in our model. For this reason we omit several nouns (the next paragraph explains why each is omitted):

- company
- office building
- display
- graphical user interface (GUI)
- user of our application
- energy
- capacity
- audio
- elevator music

We do not need to model “company,” as a class, because the company is not part of the simulation; the company simply wants us to model the elevator. We do not model the office building, or the actual place the elevator is situated, because the building does not affect how our elevator simulation operates. The phrases “display,” “audio” and “elevator music” pertain to the presentation of the model, but do not pertain to the model itself. We use these phrases when we construct the presentation in Section 22.9 and Appendix I. The phrases “graphical user interface,” “user of our application” and “First Floor and Second Floor GUI buttons” pertain to how the user controls the model, but they do not represent the model. We use these phrases when we construct the user interface in Section 12.16. “Capacity” is a property of the elevator and of the floor—not a separate entity itself. Lastly, although we’ll be saving energy with the policy of not moving the elevator until requested, we do not model “energy.”

We determine the classes for our system by grouping the remaining nouns into categories. We discard “elevator system” for the time being—we focus on designing only the system’s model and disregard how this model relates to the system as a whole. (We discuss the system as a whole in Section 13.17.) Using this logic, we discard “simulation,” because the simulation is the system in our case study. Lastly, we combine “elevator” and “elevator car” into “elevator,” because the problem statement uses the two words interchangeably.

Each remaining noun from Fig. 3.19 refers to one or more of the following categories:
• model
• elevator shaft
• elevator
• person
• floor (first floor, second floor)
• elevator door
• floor door
• elevator button
• floor button
• bell
• light

These categories are likely to be classes we will need to implement our system. Notice that we create one category for the buttons on the floors and one category for the button on the elevator. The two types of buttons perform different duties in our simulation—the buttons on the floors summon the elevator, and the button in the elevator informs the elevator to move to the other floor.

We can now model the classes in our system based on the categories we created. By convention, we capitalize class names in the design process (as we will do when we write the actual Java program that implements our design). If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., Multiple-WordName). Using this convention, we create classes ElevatorModel, ElevatorShaft, Elevator, Person, Floor, ElevatorDoor, FloorDoor, ElevatorButton, FloorButton, Bell and Light. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Class Diagrams
The UML enables us to model, via the class diagram, the classes in the elevator system and their interrelationships. Class diagrams model the structure of the system by providing the classes, or “building blocks,” of the system. Figure 3.20 represents class Elevator using the UML. In a class diagram, each class is modeled as a rectangle. We then divide this rectangle into three parts. The top part contains the name of the class. The middle part contains the class’ attributes. (We discuss attributes in “Thinking About Objects” Section 4.14 and Section 5.11.) The bottom part of the rectangle contains the class’ operations (discussed in “Thinking About Objects,” Section 6.17).

3. When we refer to the “elevator model,” we imply all classes composing the model describing the operation of our elevator system—in other words, in our simulation, several classes comprise the model. We will see in Section 13.17 that our system requires a single class to represent the model—we create class ElevatorModel to act as the “representative” for the model, because, as we will see in Fig. 3.23, ElevatorModel is the class that aggregates all other classes comprising the model.
Classes relate to one another via associations. Figure 3.21 shows how our classes `ElevatorShaft`, `Elevator` and `FloorButton` relate to one another. Notice that the rectangles in this diagram are not subdivided into three sections. The UML allows the suppression of class attributes and operations in this manner to create more readable diagrams. Such a diagram is said to be an *elided diagram*, (i.e., some information, such as the contents for the second and bottom compartments), is not modeled. We place information in these compartments in Section 4.14 and Section 6.17, respectively.

In this class diagram, a solid line that connects classes represents an *association*. An association is a relationship between classes. The numbers near the lines express *multiplicity* values. Multiplicity values indicate how many objects of a class participate in the association. From the diagram, we see that two objects of class `FloorButton` participate in the association with one object of class `ElevatorShaft`, because the two `FloorButtons` are located on the `ElevatorShaft`. Therefore, class `FloorButton` has a *two-to-one* relationship with class `ElevatorShaft`; we can also say that class `ElevatorShaft` has a *one-to-two* relationship with class `FloorButton`. We also see that class `ElevatorShaft` has a *one-to-one* relationship with class `Elevator` and vice versa. Using the UML, we can model many types of multiplicity. Figure 3.22 shows the multiplicity types and how to represent them.

An association can be named. For example, the word *Requests* above the line connecting classes `FloorButton` and `Elevator` indicates the name of that association—the arrow shows the direction of the association. This part of the diagram reads “one object of class `FloorButton` requests one object of class `Elevator`.” Note that associations are directional with the direction indicated by the arrowhead next to the association name—so it would be improper, for example, to read the preceding association as “one object of class `Elevator` requests one object of class `FloorButton`.” In addition, the word * Resets* indicates that “one object of class `ElevatorShaft` resets two objects of class `FloorButton`.” Lastly, the phrase *Signals arrival* indicates that “one object of class `Elevator` signals the `Elevator` object’s arrival to one object of class `ElevatorShaft`.”

The diamond attached to the association lines of class `ElevatorShaft` indicates that class `ElevatorShaft` has an *aggregation* relationship with classes `FloorButton` and `Elevator`. Aggregation implies a whole/part relationship. The class that has the aggregation symbol (the hollow diamond) on its end of an association line is the whole (in this case, `ElevatorShaft`), and the class on the other end of the association line is the part (in this case, classes `FloorButton` and `Elevator`). In this example, the elevator shaft “has an” elevator and two floor buttons. The “has a/has an” relationship defines aggregation (we will see in Section 9.23 that the “is a/is an” relationship defines inheritance).
Figure 3.23 shows the complete class diagram for the elevator model. We model all classes that we created, as well as the associations between these classes. [Note: In Chapter 9, we expand our class diagram by using the object-oriented concept of inheritance.]

Class **ElevatorModel** is represented near the top of the diagram and aggregates one object of class **ElevatorShaft** and two objects of class **Floor**. The **ElevatorShaft** class is an aggregation of one object of class **Elevator** and two objects each of classes **Light**, **FloorDoor** and **FloorButton**. (Notice the two-to-one relationships between each of these classes and **ElevatorShaft**.) Class **Elevator** is an aggregation of classes **ElevatorDoor**, **ElevatorButton** and **Bell**. Class **Person** has associations with both **FloorButton** and **ElevatorButton** (and other classes, as we will soon see). The association name **Presses** and the name-direction arrowheads indicate that the object of class **Person** presses these buttons. The object of class **Person** also rides the object of class **Elevator** and walks across the object of class **Floor**. The name **Requests** indicates that an object of class **FloorButton** requests the object of class **Elevator**. The name **Signals to move** indicates that the object of class **ElevatorButton** signals the object of class **Elevator** to move to the other floor. The diagram indicates many other associations, as well.
Object Diagrams

The UML also defines object diagrams, which are similar to class diagrams in that both diagrams model the structure of the system. However, object diagrams model objects (instances of classes) at a specific time in program execution. Object diagrams present a snapshot of the structure while the system is running, providing information about which objects are participating in the system at a definite point in time. Object diagrams represent relationships between objects as solid lines—these relationship are called links.

Figure 3.24 models a snapshot of the system when no people are in the building (i.e., no objects of class Person exist in the system at this point in time). Objects usually are written in the form objectName : ClassName—objectName refers to the name of the object, and ClassName refers to the class to which that object belongs. All names in an object diagram are underlined. The UML permits us to omit the object names for objects in the diagram where there exists only one object of that class (e.g., one object of class Bell at the bottom of the diagram). In large systems, object diagrams can contain many objects. This can result in cluttered, hard-to-read diagrams. If the name of a particular object is unknown, or if it is not necessary to include the name (i.e., we care only about the object type), we can disregard the object name and display only the colon and the class name.
Now we have identified the classes for our system (although we may discover others in later phases of the design process). In “Thinking About Objects,” Section 4.14, we determine the attributes for each of these classes, and in “Thinking About Objects,” Section 5.11, we use these attributes to examine how the system changes over time and to introduce its behavioral aspects. As we expand our knowledge, we will discover new information that will enable us to describe our classes more completely. Because the real world is inherently object oriented, it will be quite natural for you to pursue this project, even though you might have just begun your study of object orientation.

Questions

1. Why might it be more complicated to implement a three-story (or taller) building?
2. It is common for large buildings to have many elevators. We will see in Chapter 9 that once we have created one elevator object, it is easy to create as many as we like. What problems or opportunities do you foresee in having several elevators,
each of which may pick up and discharge passengers at every floor in a large building?

3. For simplicity, we have given our elevator and each floor a capacity of one passenger. What problems and opportunities do you foresee in being able to increase these capacities?

SUMMARY

- Applets are Java programs that can be embedded in Hypertext Markup Language (HTML) documents (i.e., Web pages). When a browser loads a Web page containing an applet, the applet downloads into the Web browser and begins execution.
- In the `appletviewer`, you can execute an applet again by clicking the appletviewer’s Applet menu and selecting the Reload option from the menu. To terminate an applet, click the appletviewer’s Applet menu and select the Quit option.
- Class `Graphics` is located in package `java.awt`. Import the `Graphics` class so the program can draw graphics.
- Class `JApplet` is located in package `javax.swing`. When you create an applet in Java, you must import the `JApplet` class.
- Java uses inheritance to create new classes from existing class definitions. Keyword `extends` followed by a class name indicates the class from which a new class inherits.
- In the inheritance relationship, the class following `extends` is called the superclass or base class and the new class is called the subclass or derived class. Using inheritance results in a new class definition that has the attributes (data) and behaviors (methods) of the superclass as well as the new features added in the subclass definition.
- A benefit of extending class `JApplet` is that someone else already has defined “what it means to be an applet.” The `appletviewer` and World Wide Web browsers that support applets expect every Java applet to have certain capabilities (attributes and behaviors), and class `JApplet` already provides those capabilities.
- Classes are used as “templates” or “blueprints” to instantiate (or create) objects in memory for use in a program. An object (or instance) is a region in the computer’s memory in which information is stored for use by the program. The term object normally implies that attributes (data) and behaviors (methods) are associated with the object and that those behaviors perform operations on the attributes of the object.
- Method `paint` is one of three methods (behaviors) that an applet container calls when any applet begins execution. These three methods are `init`, `start` and `paint`, and they are guaranteed to be called in that order.
- The parameter list is where methods receive data required to complete their tasks. Normally, this data is passed by the programmer to the method through a method call (also known as invoking a method). In the case of method `paint`, the applet container calls the method and passes the `Graphics` argument.
- Method `drawString` of class `Graphics` draws a string at the specified location on the applet. The first argument to `drawString` is the `String` to draw. The last two arguments in the list are the coordinates (or position) at which the string should be drawn. Coordinates are measured from the upper-left (0, 0) coordinate of the applet in pixels.
- You must create an HTML (Hypertext Markup Language) file to load an applet into an applet container, so the applet container can execute the applet.
Many HTML codes (referred to as tags) come in pairs. HTML tags begin with a left angle bracket < and end with a right angle bracket >.

Normally, the applet and its corresponding HTML file are stored in the same directory on disk.

The first component of the <applet> tag indicates the file containing the compiled applet class. The second and third components of the <applet> tag indicate the width and the height of the applet in pixels. Generally, each applet should be less than 800 pixels wide and 600 pixels tall.

The appletviewer only understands the <applet> and </applet> HTML tags, so it is sometimes referred to as the “minimal browser.” It ignores all other HTML tags.

Method drawLine of class Graphics draws lines. The method requires four arguments representing the two end points of the line on the applet—the x-coordinate and y-coordinate of the first end point in the line and the x-coordinate and y-coordinate of the second end point in the line. All coordinate values are specified with respect to the upper-left corner (0, 0) coordinate of the applet.

Primitive data type double stores double-precision floating-point numbers. Primitive data type float stores single-precision floating-point numbers. A double requires more memory to store a floating-point value, but stores it with approximately twice the precision of a float (15 significant digits for double vs. seven significant digits for float).

The import statements are not required if you always use the complete name of a class, including the full package name and class name (e.g., java.awt.Graphics).

The asterisk (*) notation after a package name in an import indicates that all classes in the package should be available to the compiler so the compiler can ensure that the classes are used correctly. This allows programmers to use the shorthand name (the class name by itself) of any class from the package in the program.

Every instance (object) of a class contains one copy of each of that class’s instance variables. Instance variables are declared in the body of a class definition, but not in the body of any method of that class definition. An important benefit of instance variables is that their identifiers can be used in all methods of the class.

Variables defined in the body of a method are known as local variables and can be used only in the body of the method in which they are defined.

Instance variables are always assigned a default value, and local variables are not.

Method init normally initializes the applet’s instance variables (if they need to be initialized to a value other than their default value) and performs any tasks that should occur only once when the applet begins execution.

There are actually two types of variables in Java—primitive data type variables and references.

References refer to objects in a program. References actually contain the location in the computer’s memory of an object. A reference is used to send messages to (i.e., call methods on) the object in memory. As part of the message (method call), we provide the data (arguments) that the method requires to do its task.

A variable is similar to an object. The primary difference between a variable and an object is that an object is defined by a class definition that can contain both data (instance variables) and methods, whereas a variable is defined by a primitive (or built-in) data type (one of char, byte, short, int, long, float, double or boolean) that can contain only data.

A variable can store exactly one value at a time, whereas one object can contain many individual data members.

If the data type used to declare a variable is a class name, the identifier is a reference to an object and that reference can be used to send messages to (call methods on) that object. If the data type used to declare a variable is one of the primitive data types, the identifier is a variable that can be used to store in memory or retrieve from memory a single value of the declared primitive type.
• Method `Double.parseDouble` (a static method of class `Double`) converts its `String` argument to a `double` floating-point value. Class `Double` is part of the package `java.lang`.

• Method `drawRect` draws a rectangle based on its four arguments. The first two integer values represent the upper-left x-coordinate and upper-left y-coordinate where the `Graphics` object begins drawing the rectangle. The third and fourth arguments are non-negative integers that represent the width of the rectangle in pixels and the height of the rectangle in pixels, respectively.

• To use the features of Java 2 in an applet, Sun provides the Java Plug-in to bypass a browser’s Java support and use a complete version of the Java 2 Runtime Environment (J2RE) that is installed on the user’s local computer.

• To specify that an applet should use the Java Plug-in rather than the browser’s Java support, use the HTML Converter to convert the applet’s `<applet>` and `</applet>` tags in the HTML file to indicate that the applet container should use the Plug-in to execute the applet. Sun provides the Java Plug-in 1.3 HTML Converter to perform the conversion for you.

**TERMINOLOGY**

- `applet`
- `applet container`
- `<applet>` tag
- `Applet` menu
- `appletviewer`
- `boolean` primitive type
- `browser`
- `built-in data type`
- `byte` primitive type
- `char` primitive type
- `command-line argument`
- `coordinate`
- `create an object`
- `derived class`
- `double` primitive data type
- `Double.parseDouble` method
- `double-precision floating-point number`
- `drawLine` method of class `Graphics`
- `drawRect` method of class `Graphics`
- `drawString` method of class `Graphics`
- `extends` keyword
- `float` primitive type
- `floating-point number`
- `Graphics` class
- `height` of an applet
- `HTML` Converter
- `HTML` tag
- Hypertext Markup Language (HTML)
- `import statement`
- `information hiding`
- `init` method of class `JApplet`
- `instance variable`
- `instantiate an object`
- `int` primitive type
- `interface`
- `invoke a method`
- `JApplet` class
- `java.awt` package
- `Java Plug-in`
- `Java 2 Runtime Environment (J2RE)`
- `javax.swing` package
- `local variable`
- `logic error`
- `long` primitive type
- `message`
- `method call`
- Microsoft Internet Explorer
- Netscape Communicator
- `object`
- `paint` method of class `JApplet`
- `parameter list`
- `pixel (picture element)`
- `primitive data type`
- `Quit` menu item
- `references`
- `Reload` menu item
- `short` primitive type
- `single-precision floating-point number`
- `source code`
- `start` method of class `JApplet`
- `subclass`
- `superclass`
- `text file`
- `width` of an applet
- World Wide Web
SELF-REVIEW EXERCISES

3.1  Fill in the blanks in each of the following.
    a) Class ________ provides methods for drawing.
    b) Java applets begin execution with a series of three method calls: ________, ________, and ________.
    c) Methods ________ and ________ display lines and rectangles.
    d) Keyword ________ indicates that a new class is a subclass of an existing class.
    e) Every Java applet should extend either class ________ or class ________.
    f) Java’s eight primitive data types are ________, ________, ________, ________, ________, ________, ________, and ________.

3.2  State whether each of the following is true or false. If false, explain why.
    a) To draw a rectangle, method ________ requires four arguments that specify two points on the applet.
    b) Method ________ requires four arguments that specify two points on the applet to draw a line.
    c) Type ________ is a primitive data type.
    d) Data type ________ is used to declare a floating-point number.
    e) Method ________ converts a ________ to a primitive ________ value.

3.3  Write Java statements to accomplish each of the following:
    a) Display a dialog asking the user to enter a floating-point number.
    b) Convert a ________ to a floating-point number and store the converted value in ________ variable ________.
    c) Draw the message "This is a Java program" on one line on an applet (assume you are defining this statement in the applet’s ________ method) at position (10, 10).
    d) Draw the message "This is a Java program" on two lines on an applet (assume these statements are defined in applet method ________) starting at position (10, 10) and where the first line ends with ________. Make the two lines start at the same x coordinate.

ANSWERS TO SELF-REVIEW EXERCISES

3.1  a) Graphics.  b) init, start, paint.  c) drawLine, drawRect.  d) extends.
     e) JApplet, Applet.  f) byte, short, int, long, float, double, char and boolean.

3.2  a) False. Method ________ requires four arguments—two that specify the upper-left corner of the rectangle and two that specify the width and height of the rectangle.  b) True.  c) False. Type ________ is a class in the java.lang package. Remember that names that start with a capital letter are normally class names.  d) False. Data type ________ or data type ________ can be used to declare a floating-point number. Data type ________ is used to declare integers.  e) True.

3.3  a) stringValue = JOptionPane.showInputDialog( "Enter a floating-point number" );
     b) age = Double.parseDouble( stringValue );
     c) g.drawString( "This is a Java program", 10, 10 );
     d) g.drawString( "This is a Java", 10, 10 );
     g.drawString( "program", 10, 25 );

EXERCISES

3.4  Fill in the blanks in each of the following:
    a) Data type ________ declares a single-precision floating-point variable.
b) If class `Double` provides method `parseDouble` to convert a `String` to a `double` and class `Integer` provides method `parseInt` to convert a `String` to an `int`, then class `Float` probably provides method ________ to convert a `String` to a `float`.
c) Data type ________ declares a double-precision floating-point variable.
d) The ________ or a browser can be used to execute a Java applet.
e) To load an applet into a browser you must first define a(n) ________ file.
f) The ________ and ________ HTML tags specify that an applet should be loaded into an applet container and executed.

3.5 State whether each of the following is true or false. If false, explain why.

a) All browsers support Java 2.
b) When using an `import` of the form `javax.swing.*`, all classes in the package are imported.
c) You do not need import statements if the full package name and class name are specified each time you refer to a class in a program.

3.6 Write an applet that asks the user to enter two floating-point numbers, obtains the two numbers from the user and draws the sum, product (multiplication), difference and quotient (division) of the two numbers. Use the techniques shown in Fig. 3.12.

3.7 Write an applet that asks the user to enter two floating-point numbers, obtains the numbers from the user and displays the larger number followed by the words “is larger” as a string on the applet. If the numbers are equal, print the message “These numbers are equal.” Use the techniques shown in Fig. 3.12.

3.8 Write an applet that inputs three floating-point numbers from the user and displays the sum, average, product, smallest and largest of these numbers as strings on the applet. Use the techniques shown in Fig. 3.12.

3.9 Write an applet that inputs from the user the radius of a circle as a floating-point number and draws the circle’s diameter, circumference and area. Use the value 3.14159 for \( \pi \). Use the techniques shown in Fig. 3.12. [Note: You may also use the predefined constant `Math.PI` for the value of \( \pi \). This constant is more precise than the value 3.14159. Class `Math` is defined in the `java.lang` package, so you do not need to import it.] Use the following formulas (\( r \) is the radius):

\[
\begin{align*}
diameter &= 2r \\
circumference &= 2\pi r \\
area &= \pi r^2
\end{align*}
\]

3.10 Write an applet that draws a box, an oval, an arrow and a diamond using asterisks (*) as follows:

```
**********      ***      *
*     *      * ***      *
*     *      *     *    * ***
*     *      * ***      *     *
*     *      *     *    * ***
*     *      * ***      *     *
*     *      * ***      *     *
**********      ***      *
```

3.11 Write an applet that reads five integers and determines and prints the largest and smallest integers in the group. Use only the programming techniques you learned in this chapter and Chapter 2. Draw the results on the applet.
3.12 Write an applet that reads in two floating-point numbers and determines and prints if the first is a multiple of the second. (Hint: Use the modulus operator.) Use only the programming techniques you learned in this chapter and Chapter 2. Draw the results on the applet.

3.13 Write an applet that draws a checkerboard pattern as follows:

```
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

3.14 Write an applet that draws a variety of rectangles of different sizes and locations.

3.15 Write an applet that allows the user to input the four arguments required by method `drawRect`, then draws a rectangle using the four input values.

3.16 The `Graphics` class contains a `drawOval` method that takes the same four arguments as method `drawRect`. However, the arguments for method `drawOval` specify the “bounding box” for the oval. The sides of the bounding box are the boundaries of the oval. Write a Java applet that draws an oval and a rectangle with the same four arguments. You will see that the oval touches the rectangle at the center of each side.

3.17 Modify the solution to Exercise 3.16 to output a variety of ovals of different shapes and sizes.

3.18 Write an applet that allows the user to input the four arguments required by method `drawOval`, then draws an oval using the four input values.

3.19 What does the following code print?

```java
    g.drawString( "***", 25, 25 );
    g.drawString( "****", 25, 55 );
    g.drawString( "*****", 25, 85 );
    g.drawString( "******", 25, 70 );
    g.drawString( "*******", 25, 40 );
```

3.20 Using only programming techniques from Chapter 2 and Chapter 3, write an applet that calculates the squares and cubes of the numbers from 0 to 10 and draws the resulting values in table format as follows:

<table>
<thead>
<tr>
<th>number</th>
<th>square</th>
<th>cube</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>216</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>343</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>729</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
</tbody>
</table>

[Note: This program does not require any input from the user.]
Control Structures:
Part 1

Objectives

• To understand basic problem-solving techniques.
• To be able to develop algorithms through the process of top-down, stepwise refinement.
• To be able to use the if and if/else selection structures to choose among alternative actions.
• To be able to use the while repetition structure to execute statements in a program repeatedly.
• To understand counter-controlled repetition and sentinel-controlled repetition.
• To be able to use the increment, decrement and assignment operators.

Let’s all move one place on.
Lewis Carroll
The wheel is come full circle.
William Shakespeare, King Lear
How many apples fell on Newton’s head before he took the hint!
Robert Frost, comment
4.1 Introduction

Before writing a program to solve a problem, it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem. When writing a program, it is equally essential to understand the types of building blocks that are available and to employ proven program construction principles. In this chapter and in Chapter 5, we discuss these issues in our presentation of the theory and principles of structured programming. The techniques you learn here are applicable to most high-level languages, including Java. When we study object-based programming in more depth in Chapter 8, we will see that control structures are helpful in building and manipulating objects.

4.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of

1. the actions to be executed and
2. the order in which the actions are to be executed

is called an algorithm. The following example demonstrates that correctly specifying the order in which the actions are to be executed is important.
Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work.

This routine gets the executive to work well prepared to make critical decisions. Suppose, however, that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work.

In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements are to be executed in a computer program is called program control. In this chapter and Chapter 5, we investigate the program control capabilities of Java.

### 4.3 Pseudocode

Pseudocode is an artificial and informal language that helps programmers develop algorithms. The pseudocode we present here is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday English; it is convenient and user friendly, although it is not an actual computer programming language.

Pseudocode programs are not actually executed on computers. Rather, they help the programmer “think out” a program before attempting to write it in a programming language, such as Java. In this chapter, we give several examples of pseudocode programs.

**Software Engineering Observation 4.1**

Pseudocode is often used to “think out” a program during the program design process. Then the pseudocode program is converted to Java.

The style of pseudocode we present consists purely of characters, so programmers may conveniently type pseudocode programs using an editor program. The computer can produce a freshly printed copy of a pseudocode program on demand. A carefully prepared pseudocode program may be converted easily to a corresponding Java program. This conversion is done in many cases simply by replacing pseudocode statements with their Java equivalents.

Pseudocode normally describes only executable statements—the actions that are performed when the program is converted from pseudocode to Java and is run. Declarations are not executable statements. For example, the declaration

```java
int i;
```

tells the compiler the type of variable `i` and instructs the compiler to reserve space in memory for the variable. This declaration does not cause any action—such as input, output or a calculation—to occur when the program is executed. Some programmers choose to list variables and mention the purpose of each at the beginning of a pseudocode program.

### 4.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they are written. This process is called sequential execution. Various Java statements we will soon discuss enable the programmer to specify that the next statement to be executed may be other than the next one in sequence. This is called transfer of control.
During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The finger of blame was pointed at the \textit{goto statement} (used in several programming languages, including C and Basic), which allows the programmer to specify a transfer of control to one of a very wide range of possible destinations in a program. The notion of so-called \textit{structured programming} became almost synonymous with “\textit{goto} elimination.” Java does not have a \textit{goto} statement; however, \textit{goto} is a reserved word and should not be used in a Java program.

The research of Bohm and Jacopini\(^1\) had demonstrated that programs could be written without any \textit{goto} statements. The challenge of the era for programmers was to shift their styles to “\textit{goto}-less programming.” It was not until the 1970s that programmers started taking structured programming seriously. The results have been impressive, as software development groups have reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes is that structured programs are clearer, easier to debug and modify and more likely to be bug free in the first place.

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three \textit{control structures}—namely, the \textit{sequence structure}, the \textit{selection structure} and the \textit{repetition structure}. The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they are written. The \textit{flowchart} segment in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order.

A flowchart is a graphical representation of an algorithm or a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called \textit{flowlines}, which indicate the order in which the actions of the algorithm execute.

Like pseudocode, flowcharts are often useful for developing and representing algorithms, although pseudocode is strongly preferred by many programmers. Flowcharts show clearly how control structures operate; that is all we use them for in this text. The reader should carefully compare the pseudocode and flowchart representations of each control structure.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{sequence_structure_flowchart.png}
\caption{Flowcharting Java’s sequence structure.}
\end{figure}


© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
Consider the flowchart segment for the sequence structure in Fig. 4.1. We use the rectangle symbol, also called the action symbol, to indicate any type of action, including a calculation or an input/output operation. The flowlines in the figure indicate the order in which the actions are to be performed; first, grade is to be added to total, and then 1 is to be added to counter. Java allows us to have as many actions as we want in a sequence structure. As we will soon see, anywhere a single action may be placed, we may instead place several actions in sequence.

When drawing a flowchart that represents a complete algorithm, an oval symbol containing the word “Begin” is the first symbol used in the flowchart; an oval symbol containing the word “End” indicates where the algorithm ends. When drawing only a portion of an algorithm, as in Fig. 4.1, the oval symbols are omitted in favor of small circle symbols, also called connector symbols.

Perhaps the most important flowcharting symbol is the diamond symbol, also called the decision symbol, which indicates that a decision is to be made. We will discuss the diamond symbol in the next section.

Java provides three types of selection structures; we discuss each in this chapter and in Chapter 5. The if selection structure either performs (selects) an action, if a condition is true, or skips the action, if the condition is false. The if/else selection structure performs an action if a condition is true and performs a different action if the condition is false. The switch selection structure (Chapter 5) performs one of many different actions, depending on the value of an expression.

The if structure is called a single-selection structure, because it selects or ignores a single action (or, as we will soon see, a single group of actions). The if/else structure is called a double-selection structure, because it selects between two different actions (or groups of actions). The switch structure is called a multiple-selection structure, because it selects among many different actions (or groups of actions).

Java provides three types of repetition structures—namely, while, do/while and for. (do/while and for are covered in Chapter 5.) Each of the words if, else, switch, while, do and for are Java keywords. These words are reserved by the language to implement various features, such as Java’s control structures. Keywords cannot be used as identifiers, such as for variable names. A complete list of Java keywords is shown in Fig. 4.2.

<table>
<thead>
<tr>
<th>Java Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
</tr>
<tr>
<td>catch</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>final</td>
</tr>
<tr>
<td>implements</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>private</td>
</tr>
</tbody>
</table>

Fig. 4.2  Java keywords (part 1 of 2).
Well, that is all there is. Java has only seven control structures: the sequence structure, three types of selection structures and three types of repetition structures. Each program is formed by combining as many of each type of control structure as is appropriate for the algorithm the program implements. As with the sequence structure in Fig. 4.1, we will see that each control structure is flowcharted with two small circle symbols, one at the entry point to the control structure and one at the exit point.

Single-entry/single-exit control structures make it easy to build programs; the control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This procedure is similar to the way in which a child stacks building blocks, so we call it control-structure stacking. We will learn that there is only one other way in which control structures may be connected: control-structure nesting. Thus, algorithms in Java programs are constructed from only seven different types of control structures, combined in only two ways.

### 4.5 The if Selection Structure

A selection structure is used to choose among alternative courses of action in a program. For example, suppose that the passing grade on an examination is 60 (out of 100). Then the pseudocode statement

\[
\text{If student’s grade is greater than or equal to 60}
\]

\[
\text{Print “Passed”}
\]

determines if the condition “student’s grade is greater than or equal to 60” is true or false. If the condition is true, then “Passed” is printed, and the next pseudocode statement in order is “performed.” (Remember that pseudocode is not a real programming language.) If the condition is false, the Print statement is ignored, and the next pseudocode statement in order is performed. Note that the second line of this selection structure is indented. Such indentation is optional, but it is highly recommended, because it emphasizes the inherent structure of structured programs. The Java compiler ignores white-space characters, like blanks, tabs and newlines, used for indentation and vertical spacing. Programmers insert these white-space characters to enhance program clarity.

---

**Java Keywords**

<table>
<thead>
<tr>
<th>static</th>
<th>super</th>
<th>switch</th>
<th>synchronized</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>throw</td>
<td>throws</td>
<td>transient</td>
<td>true</td>
<td>try</td>
</tr>
<tr>
<td>void</td>
<td>volatile</td>
<td>while</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Keywords that are reserved, but not used, by Java*

const  goto

---

**Common Programming Error 4.1**

*Using a keyword as an identifier is a syntax error.*
**Good Programming Practice 4.1**

Consistently applying reasonable indentation conventions throughout your programs improves program readability. We suggest a fixed-size tab of about $\frac{1}{4}$ inch or three spaces per indent.

The preceding pseudocode if statement may be written in Java as

```java
if ( studentGrade >= 60 )
    System.out.println( "Passed" );
```

Notice that the Java code corresponds closely to the pseudocode. This attribute is a property of pseudocode that makes it a useful program development tool. The statement in the body of the if structure outputs the character string "Passed" in the command window.

The flowchart in Fig. 4.3 illustrates the single-selection if structure. This flowchart contains what is perhaps the most important flowcharting symbol—the diamond symbol, also called the decision symbol, which indicates that a decision is to be made. The decision symbol contains an expression, such as a condition, that can be either true or false. The decision symbol has two flowlines emerging from it. One indicates the direction to be taken when the expression in the symbol is true; the other indicates the direction to be taken when the expression is false. A decision can be made on any expression that evaluates to a value of Java's boolean type (i.e., any expression that evaluates to true or false).

Note that the if structure is a single-entry/single-exit structure. We will soon learn that the flowcharts for the remaining control structures also contain (besides small circle symbols and flowlines) only rectangle symbols, to indicate the actions to be performed, and diamond symbols, to indicate decisions to be made. This factor is indicative of the action/decision model of programming we have been emphasizing throughout this chapter.

We can envision seven bins, each containing only control structures of one of the seven types. These control structures are empty; nothing is written in the rectangles or in the diamonds. The programmer’s task, then, is to assemble a program from as many of each type of control structure as the algorithm demands, combining the control structures in only two possible ways (stacking or nesting) and then filling in the actions and decisions in a manner appropriate for the algorithm. In this chapter we discuss the variety of ways in which actions and decisions may be written.

![Flowchart of single-selection if structure](image-url)
4.6 The if/else Selection Structure

The if selection structure performs an indicated action only when the given condition evaluates to true; otherwise, the action is skipped. The if/else selection structure allows the programmer to specify that a different action is to be performed when the condition is true rather than when the condition is false. For example, the pseudocode statement

\[
\begin{align*}
\text{If student's grade is greater than or equal to 60} \\
\text{Print “Passed”} \\
\text{else} \\
\text{Print “Failed”}
\end{align*}
\]

prints Passed if the student’s grade is greater than or equal to 60 and prints Failed if the student’s grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is “performed.” Note that the body of the else is also indented.

**Good Programming Practice 4.2**

Indent both body statements of an if/else structure.

The indentation convention you choose should be carefully applied throughout your programs. It is difficult to read programs that do not use uniform spacing conventions.

The preceding pseudocode If/else structure may be written in Java as

```java
if (studentGrade >= 60)
    System.out.println( "Passed" );
else
    System.out.println( "Failed" );
```

The flowchart in Fig. 4.4 nicely illustrates the flow of control in an if/else structure. Once again, note that, besides small circles and arrows, the only symbols in the flowchart are rectangles (for actions) and a diamond (for a decision). We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty double-selection structures as might be needed to build a Java algorithm. The programmer’s job is to assemble the selection structures (by stacking and nesting) with other control structures required by the algorithm and to fill in the empty rectangles and empty diamonds with actions and decisions appropriate to the algorithm being implemented.

**Fig. 4.4** Flowcharting the double-selection if/else structure.
The conditional operator (?:) is related to the if/else structure. ?: is Java’s only ternary operator—it takes three operands. The operands together with ?: form a conditional expression. The first operand is a boolean expression, the second is the value for the conditional expression if the condition evaluates to true and the third is the value for the conditional expression if the condition evaluates to false. For example, the statement

```java
System.out.println( studentGrade >= 60 ? "Passed" : "Failed" );
```
contains a conditional expression that evaluates to the string "Passed" if the condition `studentGrade >= 60` is true and to the string "Failed" if the condition is false. Thus, this statement with the conditional operator performs essentially the same function as the if/else statement given previously. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses. We will see that conditional operators can be used in some situations where if/else statements cannot.

**Good Programming Practice 4.3**

In general, conditional expressions are more difficult to read than if/else structures. Such expressions should be used with discretion when they help improve a program’s readability.

Nested if/else structures test for multiple cases by placing if/else structures inside if/else structures. For example, the following pseudocode statement prints A for exam grades greater than or equal to 90, B for grades in the range 80 to 89, C for grades in the range 70 to 79, D for grades in the range 60 to 69, and F for all other grades:

```plaintext
If student’s grade is greater than or equal to 90
Print “A”
else
  If student’s grade is greater than or equal to 80
    Print “B”
  else
    If student’s grade is greater than or equal to 70
      Print “C”
    else
      If student’s grade is greater than or equal to 60
        Print “D”
      else
        Print “F”
```

This pseudocode may be written in Java as

```java
if ( studentGrade >= 90 )
  System.out.println( "A" );
else
  if ( studentGrade >= 80 )
    System.out.println( "B" );
  else
    if ( studentGrade >= 70 )
      System.out.println( "C" );
    else
      if ( studentGrade >= 60 )
        System.out.println( "D" );
      else
        System.out.println( "F" );
```

7/2/01
If `studentGrade` is greater than or equal to 90, the first four conditions will be true, but only the `System.out.println` statement after the first test will be executed. After that particular `System.out.println` is executed, the `else` part of the “outer” `if/else` statement is skipped.

### Good Programming Practice 4.4

> If there are several levels of indentation, each level should be indented by the same additional amount of space.

Most Java programmers prefer to write the preceding `if` structure as

```java
if ( grade >= 90 )
    System.out.println( "A" );
else if ( grade >= 80 )
    System.out.println( "B" );
else if ( grade >= 70 )
    System.out.println( "C" );
else if ( grade >= 60 )
    System.out.println( "D" );
else
    System.out.println( "F" );
```

Both forms are equivalent. The latter form is popular because it avoids the deep indentation of the code to the right. Such deep indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

It is important to note that the Java compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces (`{}`). This attribute is referred to as the *dangling-else problem*. For example,

```java
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
else
    System.out.println( "x is <= 5" );
```

Appears to indicate that if `x` is greater than 5, the `if` structure in its body determines if `y` is also greater than 5. If so, the string "*x and y are > 5"* is output. Otherwise, it appears that if `x` is not greater than 5, the `else` part of the `if/else` structure outputs the string "*x is <= 5"*.

Beware! The preceding nested `if` structure does not execute as it would appear to. The compiler actually interprets the preceding structure as

```java
if ( x > 5 )
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
else
    System.out.println( "x is <= 5" );
```

in which the body of the first `if` structure is an `if/else` structure. This structure tests if `x` is greater than 5. If so, execution continues by testing if `y` is also greater than 5. If the second condition is true, the proper string—"*x and y are > 5"*—is displayed. However, if the second condition is false, the string "*x is <= 5"* is displayed, even though we know that `x` is greater than 5.
To force the preceding nested if structure to execute as it was originally intended, the structure must be written as follows:

```java
if ( x > 5 ) {
    if ( y > 5 )
        System.out.println( "x and y are > 5" );
} else
    System.out.println( "x is <= 5" );
```

The braces ({})) indicate to the compiler that the second if structure is in the body of the first if structure and that the else is matched with the first if structure. In Exercise 4.21 and Exercise 4.22, you will investigate the dangling-else problem further.

The if selection structure normally expects only one statement in its body. To include several statements in the body of an if structure, enclose the statements in braces ({ and }). A set of statements contained within a pair of braces is called a block.

**Software Engineering Observation 4.2**

A block can be placed anywhere in a program that a single statement can be placed.

The following example includes a block in the else part of an if/else structure:

```java
if ( grade >= 60 )
    System.out.println( "Passed" );
else {
    System.out.println( "Failed" );
    System.out.println( "You must take this course again." );
}
```

In this case, if grade is less than 60, the program executes both statements in the body of the else and prints

```
Failed.
You must take this course again.
```

Notice the braces surrounding the two statements in the else clause. These braces are important. Without the braces, the statement

```
System.out.println( "You must take this course again." );
```

would be outside the body of the else part of the if structure and would execute regardless of whether the grade is less than 60.

**Common Programming Error 4.2**

Forgetting one or both of the braces that delimit a block can lead to syntax or logic errors.

Syntax errors (such as when one brace in a block is left out of the program) are caught by the compiler. A logic error (such as when both braces in a block are left out of the program) has its effect at execution time. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but the program produces incorrect results.
Software Engineering Observation 4.3

Just as a block can be placed anywhere a single statement can be placed, it is also possible to have no statement at all (i.e., the empty statement in such places). The empty statement is represented by placing a semicolon (;) where a statement would normally be.

Common Programming Error 4.3

Placing a semicolon after the condition in an `if` structure leads to a logic error in single-selection `if` structures and a syntax error in double-selection `if` structures (if the `if` part contains a nonempty body statement).

Good Programming Practice 4.5

Some programmers prefer to type the beginning and ending braces of blocks before typing the individual statements within the braces. This practice helps avoid omitting one or both of the braces.

In this section, we have introduced the notion of a block. A block may contain declarations (as does the body of `main`, for example). The declarations in a block commonly are placed first in the block before any action statements occur, but declarations may also be intermixed with action statements.

4.7 The `while` Repetition Structure

A repetition structure allows the programmer to specify that an action is to be repeated while some condition remains true. The pseudocode statement

```
While there are more items on my shopping list
    Purchase next item and cross it off my list
```

describes the repetition that occurs during a shopping trip. The condition “there are more items on my shopping list” may be true or false. If it is true, then the action “Purchase next item and cross it off my list” is performed. This action will be performed repeatedly while the condition remains true. The statement(s) contained in the `while` repetition structure constitute the body of the `while` structure. The body of the `while` structure may be a single statement or a block. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first pseudocode statement after the repetition structure is executed.

Common Programming Error 4.4

Not providing in the body of a `while` structure an action that eventually causes the condition in the `while` to become false is a logic error. Normally, such a repetition structure will never terminate—an error called an infinite loop.

Common Programming Error 4.5

Spelling the keyword `while` with an uppercase W, as in `While`, is a syntax error. (Remember that Java is a case-sensitive language.) All of Java’s reserved keywords, such as `while`, `if` and `else`, contain only lowercase letters.

As an example of a `while` structure, consider a program segment designed to find the first power of 2 larger than 1000. Suppose that the `int` variable `product` has been initialized to `2`. When the following `while` structure finishes executing, `product` contains the result:
int product = 2;

while ( product <= 1000 )
    product = 2 * product;

The flowchart in Fig. 4.5 illustrates the flow of control of the preceding while repetition structure. Once again, note that, besides small circles and arrows, the flowchart contains only a rectangle symbol and a diamond symbol.

Imagine, again, a deep bin of empty while structures that may be stacked and nested with other control structures to form a structured implementation of an algorithm’s flow of control. The empty rectangles and diamonds are then filled in with appropriate actions and decisions. The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision, which is tested each time through the loop until the decision eventually becomes false. At this point, the while structure is exited, and control passes to the next statement in the program.

When the while structure is entered, product is 2. Variable product is repeatedly multiplied by 2, taking on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024 successively. When product becomes 1024, the condition product <= 1000 in the while structure becomes false. This result terminates the repetition, with 1024 as product’s final value. Execution continues with the next statement after the while. [Note: If a while structure’s condition is initially false, the body statement(s) will never be performed.]

4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

To illustrate how algorithms are developed, we solve several variations of a class-averaging problem. Consider the following problem statement:

A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation and print the result.
Let us use pseudocode to list the actions to be executed and specify the order in which these actions should be executed. We use counter-controlled repetition to input the grades one at a time. This technique uses a variable called a counter to control the number of times a set of statements will execute. In this example, repetition terminates when the counter exceeds 10. In this section, we present a pseudocode algorithm (Fig. 4.6) and the corresponding program (Fig. 4.7) to solve this problem using counter-controlled repetition. In the next section, we show how pseudocode algorithms are developed. Counter-controlled repetition is often called definite repetition, because the number of repetitions is known before the loop begins executing.

Note the references in the algorithm to a total and a counter. A total is a variable used to accumulate the sum of a series of values. A counter is a variable used to count—in this case, to count the number of grades entered. Variables used to store totals should normally be initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total’s memory location.

---

Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

---

Fig. 4.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

---

```java
// Fig. 4.7: Average1.java
// Class average program with counter-controlled repetition.

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

public class Average1 {

    // main method begins execution of Java application
    public static void main( String args[] )
    {
        int total,          // sum of grades input by user
            gradeCounter,   // number of grades entered
            gradeValue,     // grade value
            average;        // average of all grades
        String grade;       // grade typed by user
```

---

Fig. 4.7 Class-average program with counter-controlled repetition (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
// Initialization Phase
total = 0; // clear total
gradeCounter = 1; // prepare to loop

// Processing Phase
while ( gradeCounter <= 10 ) { // loop 10 times

    // prompt for input and read grade from user
    grade = JOptionPane.showInputDialog(
        "Enter integer grade: ");

    // convert grade from a String to an integer
    gradeValue = Integer.parseInt( grade );

    // add gradeValue to total
    total = total + gradeValue;

    // add 1 to gradeCounter
    gradeCounter = gradeCounter + 1;
}

// Termination Phase
average = total / 10; // perform integer division

// display average of exam grades
JOptionPane.showMessageDialog( null,
    "Class average is " + average, "Class Average",
    JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 ); // terminate the program

} // end method main

} // end class Average1

Fig. 4.7 Class-average program with counter-controlled repetition (part 2 of 3).
Good Programming Practice 4.6

Initialize counters and totals.

Line 5,

```java
import javax.swing.JOptionPane;
```

imports class `JOptionPane` to enable the program to read data from the keyboard and output data to the screen using the input dialog and message dialog shown in Chapter 2.

Line 7 begins the definition of application class `Average1`. Remember that the definition of an application class must contain a `main` method (lines 10–49) in order for the application to be executed.

Lines 12–16,

```java
int total,              // sum of grades
    gradeCounter, // number of grades entered
    gradeValue,  // grade value
    average,     // average of all grades
    String grade; // grade typed by user
```

declare variables `total`, `gradeCounter`, `gradeValue` and `average` to be of type `int` and variable `grade` to be of type `String`. Variable `grade` stores the `String` the
user types in the input dialog. Variable \texttt{gradeValue} stores the integer value of \texttt{grade} after the program converts it from a \texttt{String} to an \texttt{int}.

Notice that the preceding declarations appear in the body of method \texttt{main}. Remember that variables declared in a method definition’s body are \textit{local variables} and can be used only from the line of their declaration in the method to the closing right brace (}) of the method definition. A local variable’s declaration must appear before the variable is used in that method. A local variable declared in one method of a class cannot be accessed directly by other methods of a class.

\textbf{Good Programming Practice 4.7}

Always place a blank line before a declaration that appears between executable statements. This format makes the declarations stand out in the program and contributes to program clarity.

\textbf{Good Programming Practice 4.8}

If you prefer to place declarations at the beginning of a method, separate the declarations from the executable statements in that method with one blank line, to highlight where the declarations end and the executable statements begin.

\textbf{Common Programming Error 4.6}

Attempting to use a local variable’s value before initializing the variable (normally with an assignment statement) results in a compile error indicating that the variable may not have been initialized. The value of a local variable cannot be used until the variable is initialized. The program will not compile properly until the variable receives an initial value.

Lines 19–20,

\begin{verbatim}
    total = 0;           // clear total
    gradeCounter = 1;   // prepare to loop
\end{verbatim}

are assignment statements that initialize \texttt{total} to 0 and \texttt{gradeCounter} to 1. Note that these statements initialize variables \texttt{total} and \texttt{gradeCounter} before they are used in calculations.

Line 23,

\begin{verbatim}
while ( gradeCounter <= 10 ) {  // loop 10 times
\end{verbatim}

indicates that the \texttt{while} structure should continue looping (also called \textit{iterating}) as long as the value of \texttt{gradeCounter} is less than or equal to 10.

Lines 26–27,

\begin{verbatim}
grade = JOptionPane.showInputDialog(  
    "Enter integer grade: " );
\end{verbatim}

correspond to the pseudocode statement “\textit{Input the next grade.}” The statement displays an input dialog with the prompt “\textit{Enter integer grade:}” on the screen.

After the user enters the \texttt{grade}, the program converts it from a \texttt{String} to an \texttt{int} at line 30,

\begin{verbatim}
gradeValue = Integer.parseInt( grade );
\end{verbatim}

Remember that class \texttt{Integer} is from package \texttt{java.lang} that the compiler imports in every Java program. The pseudocode for the class-average problem does not reflect the pre-
ceding statement. The pseudocode statement “Input the next grade” requires the programmer to implement the process of obtaining the value from the user and converting it to a type that can be used in calculating the average. As you learn to program, you will find that you require fewer pseudocode statements to help you implement a program.

Next, the program updates the total with the new gradeValue entered by the user. Line 33,

```
total = total + gradeValue;
```

adds gradeValue to the previous value of total and assigns the result to total.

Line 36,

```
gradeCounter = gradeCounter + 1;
```

adds 1 to gradeCounter to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing gradeCounter is necessary for the condition in the while structure to become false eventually and terminate the loop.

Line 41,

```
average = total / 10;  // perform integer division
```

assigns the results of the average calculation to variable average. Lines 44–46,

```
JOptionPane.showMessageDialog(  
    null, "Class average is " + average, "Class Average",  
    JOptionPane.INFORMATION_MESSAGE  );
```

display an information message dialog containing the string "Class average is " followed by the value of variable average. The string "Class Average" (the third argument) is the title of the message dialog.

Line 48,

```
System.exit( 0 );      // terminate the program
```

terminates the program.

After compiling the class definition with javac, execute the application from the command window with the command

```
java Average1
```

This command executes the Java interpreter and tells it that the main method for this application is defined in class Average1.

Note that the averaging calculation in the program produced an integer result. Actually, the sum of the grade-point values in this example is 794, which, when divided by 10, should yield 79.4 (i.e., a number with a decimal point). We will see how to deal with such numbers (called floating-point numbers) in the next section.

4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

Let us generalize the class-average problem. Consider the following problem:

*Develop a class-averaging program that processes an arbitrary number of grades each time the program executes.*
In the first class-average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will input. The program must process an arbitrary number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a sentinel value (also called a signal value, a dummy value or a flag value) to indicate the end of data entry. The user types grades in until all legitimate grades have been entered. The user then types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called indefinite repetition, because the number of repetitions is not known before the loop begins executing.

Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value. Because grades on a quiz are normally nonnegative integers, –1 is an acceptable sentinel value for this problem. Thus, an execution of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. In this case, the program would compute and print the class average for the grades 95, 96, 75, 74 and 89. (–1 is the sentinel value, so it should not enter into the averaging calculation.)

**Common Programming Error 4.7**

Choosing a sentinel value that is also a legitimate data value results in a logic error and may prevent a sentinel-controlled loop from terminating properly.

We approach the class-average program with a technique called top-down, stepwise refinement, a method that is essential to the development of well-structured algorithms. We begin with a pseudocode representation of the top:

*Determine the class average for the quiz*

The top is a single statement that conveys the overall function of the program. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail from which to write the Java algorithm. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these tasks in the order in which they need to be performed. This procedure results in the following first refinement:

*Initialize variables*

*Input, sum up and count the quiz grades*

*Calculate and print the class average*

This pseudocode uses only the sequence structure—the steps listed occur in order, one after the other.

**Software Engineering Observation 4.4**

Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

To proceed to the next level of refinement (i.e., the second refinement), we commit to specific variables. We need a running total of the grades, a count of how many grades have been processed, a variable to receive the value of each grade as it is input and a variable to store the calculated average. The pseudocode statement

*Initialize variables*
may be refined as follows:

\begin{quote}
\textit{Initialize total to zero}
\textit{Initialize counter to zero}
\end{quote}

Notice that only the variables \textit{total} and \textit{counter} are initialized before they are used; the variables \textit{average} and \textit{grade} (for the calculated average and the user input, respectively) need not be initialized, because their values are replaced as they are calculated or input.

The pseudocode statement

\begin{quote}
Input, sum up and count the quiz grades
\end{quote}

requires a repetition structure (i.e., a loop) that successively inputs each grade. We do not know how many grades the user will input, so the program will use sentinel-controlled repetition. The user at the keyboard inputs legitimate grades one at a time. After inputting the last legitimate grade, the user types the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user inputs the sentinel value. The second refinement of the preceding pseudocode statement is then

\begin{quote}
Input the first grade (possibly the sentinel)
\end{quote}

\begin{quote}
While the user has not as yet entered the sentinel
\hspace{1em}Add this grade into the running total
\hspace{1em}Add one to the grade counter
\hspace{1em}Input the next grade (possibly the sentinel)
\end{quote}

Notice that in pseudocode, we do not use braces around the pseudocode that forms the body of the \textit{while} structure. We simply indent the pseudocode under the \textit{while}, to show that it belongs to the \textit{while}. Again, pseudocode is only an informal program development aid.

The pseudocode statement

\begin{quote}
Calculate and print the class average
\end{quote}

may be refined as follows:

\begin{quote}
If the counter is not equal to zero
\hspace{1em}Set the average to the total divided by the counter
\hspace{1em}Print the average
else
\hspace{1em}Print “No grades were entered”
\end{quote}

Notice that we are testing for the possibility of division by zero—a logic error that, if undetected, would cause the program to produce invalid output. The complete second refinement of the pseudocode algorithm for the class-average problem is shown in Fig. 4.8.

\textbf{Testing and Debugging Tip 4.1}

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as by printing an error message) rather than allowing the division by zero to occur.

\textbf{Good Programming Practice 4.9}

Include completely blank lines in pseudocode programs to make the pseudocode more readable. The blank lines separate pseudocode control structures, as well as the phases of the programs.
Many algorithms can be divided logically into three phases: an initialization phase that initializes the program variables; a processing phase that inputs data values and adjusts program variables accordingly and a termination phase that calculates and displays the results.

The pseudocode algorithm in Fig. 4.8 solves the more general class-averaging problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary.

The Java application and a sample execution are shown in Fig. 4.9. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point (i.e., a real number). The type `int` cannot represent real numbers (i.e., numbers with decimal points), so this program uses data type `double` to handle floating-point numbers. The program introduces a special operator called a *cast operator* to handle the type conversion we will need for the averaging calculation. These features are explained in detail in the discussion of the application.

In this example, we see that control structures may be stacked on top of one another (in sequence) just as a child stacks building blocks. The *while* structure (lines 33–47) is followed by an *if/else* structure (lines 52–63) in sequence. Much of the code in this program is identical to the code in Fig. 4.7, so we concentrate in this example on the new features and issues.
Fig. 4.9: Average2.java

// Class average program with sentinel-controlled repetition.

// Java core packages
import java.text.DecimalFormat;

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

public class Average2 {
    public static void main( String args[] )
    {
        int gradeCounter, // number of grades entered
            gradeValue, // grade value
            total; // sum of grades
        double average; // average of all grades
        String input; // grade typed by user

        // Initialization phase
        total = 0; // clear total
        gradeCounter = 0; // prepare to loop

        // Processing phase
        // prompt for input and read grade from user
        input = JOptionPane.showInputDialog( // convert grade from a String to an integer
            "Enter Integer Grade, -1 to Quit:" );

        while ( gradeValue != -1 ) {
            total = total + gradeValue;
            gradeCounter = gradeCounter + 1;

            // prompt for input and read grade from user
            input = JOptionPane.showInputDialog;
            gradeValue = Integer.parseInt( input );

            // add 1 to gradeCounter
            gradeCounter = gradeCounter + 1;

            // convert grade from a String to an integer
            gradeValue = Integer.parseInt( input );
        }

        // Termination phase
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );

        if ( gradeCounter != 0 ) {
            average = (double) total / gradeCounter;
        }
    }
}
Line 18 declares double variable average. This change allows us to store the class average as a floating-point number. Line 23 initializes gradeCounter to 0, because no grades have been entered yet. Remember that this program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, variable gradeCounter is incremented only when the user inputs a valid grade value.

Notice the difference in program logic for sentinel-controlled repetition as compared with the counter-controlled repetition in Fig. 4.7. In counter-controlled repetition, each iteration (loop) of the while structure reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads and converts one value (lines 27–31) before reaching the while structure. This value determines whether the program’s flow of control should enter the body of the while structure. If the condition of the while structure is false, the user entered the sentinel, so the body of the while structure does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the loop adds the value input by the user to the average.
After the value has been processed, lines 42–46 in the loop’s body input the next value from the user before program control reaches the end of the `while` structure’s body. As program control reaches the closing right brace (}) of the body at line 47, execution continues with the next test of the condition of the `while` structure (line 33). The condition uses the new value just input by the user to determine if the `while` structure’s body should execute again. Notice that the next value always is input from the user immediately before the program tests the condition of the `while` structure. This structure allows the program to determine if the value just input by the user is the sentinel value before the program processes that value (i.e., adds it to the `total`). If the value input is the sentinel value, the `while` structure terminates, and the program does not add the value to the `total`.

Notice the block in the `while` loop in Fig. 4.9. Without the braces, the last four statements in the body of the loop would fall outside the loop, causing the computer to interpret the code incorrectly as follows:

```java
while ( gradeValue != -1 )
    // add gradeValue to total
    total = total + gradeValue;
    // add 1 to gradeCounter
    gradeCounter = gradeCounter + 1;
    // prompt for input and read grade from user
    input = JOptionPane.showInputDialog( "Enter Integer Grade, -1 to Quit:" );
    // convert grade from a String to an integer
    gradeValue = Integer.parseInt( input );
```

This code would cause an infinite loop in the program if the user does not input the sentinel -1 as the input value at lines 27–28 (before the `while` structure) in the program.

**Common Programming Error 4.8**

Omitting the curly braces that are needed to delineate a block can lead to logic errors such as infinite loops. To prevent this problem, some programmers enclose the body of every control structure in braces.

**Good Programming Practice 4.10**

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user of the value that represents the sentinel.

```
Line 50,

DecimalFormat twoDigits = new DecimalFormat( "0.00" );
```

declares `twoDigits` as a reference to an object of class `DecimalFormat` (package `java.text`). `DecimalFormat` objects format numbers. In this example, we want to output the class average with two digits to the right of the decimal point (i.e., rounded to the nearest hundredth). The preceding line creates a `DecimalFormat` object that is initialized with the string "0.00". Each 0 is a format flag that specifies a required digit position in the formatted floating-point number. This particular format indicates that every number formatted with `twoDigits` will have at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point. If the number does not meet the formatting requirements, 0s are inserted in the formatted number at the required positions. The `new` operator creates an
object as the program executes by obtaining enough memory to store an object of the type
specified to the right of `new`. The process of creating new objects is also known as **creating an instance**, or **instantiating an object**. Operator `new` is known as the **dynamic memory allocation operator**. The value in parentheses after the type in a `new` operation is used to **initialize** (i.e., give a value to) the new object. Reference `twoDigits` is assigned the result of the `new` operation by using the **assignment operator**, `=`. The statement is read as “`twoDigits gets the value of new DecimalFormat("0.00")`.”

**Software Engineering Observation 4.7**

Normally, objects are created with operator `new`. One exception to this is a string literal that is contained in quotes, such as “hello”. String literals are treated as objects of class `String` and are instantiated automatically.

Averages do not always evaluate to integer values. Often, an average is a value that contains a fractional part, such as 3.333 or 2.7. These values are referred to as **floating-point numbers** and are represented by the data type `double`. The variable `average` is declared to be of type `double` to capture the fractional result of our calculation. However, the result of the calculation `total / gradeCounter` is an integer, because `total` and `gradeCounter` are both integer variables. Dividing two integers results in **integer division**—any fractional part of the calculation is lost (i.e., **truncated**). The fractional part of the calculation is lost before the result can be assigned to `average`, because the calculation is performed before the assignment occurs.

To perform a floating-point calculation with integer values, we must create temporary values that are floating-point numbers for the calculation. Java provides the **unary cast operator** to accomplish this task. Line 53,

```java
average = (double) total / gradeCounter;
```

uses the cast operator `(double)` to create a temporary floating-point copy of its operand—`total`. Using a cast operator in this manner is called **explicit conversion**. The value stored in `total` is still an integer. The calculation now consists of a floating-point value (the temporary `double` version of `total`) divided by the integer `gradeCounter`. Java knows how to evaluate only arithmetic expressions in which the operands’ data types are identical. To ensure that the operands are of the same type, Java performs an operation called **promotion** (or **implicit conversion**) on selected operands. For example, in an expression containing the data types `int` and `double`, the values of `int` operands are **promoted** to `double` values for use in the expression. In this example, Java promotes the value of `gradeCounter` to type `double`, and then the program performs the calculation and assigns the result of the floating-point division to `average`. Later in this chapter, we discuss all of the standard data types and their order of promotion.

**Common Programming Error 4.9**

The cast operator can be used to convert between primitive numeric types and to convert between related class types (as we discuss in Chapter 9). Casting a variable to the wrong type may cause compilation errors or runtime errors.

Cast operators are available for any data type. The cast operator is formed by placing parentheses around the name of a data type. The operator is a **unary operator** (i.e., an operator that takes only one operand). In Chapter 2, we studied the binary arithmetic operators. Java also supports unary versions of the plus (`+`) and minus (`-`) operators, so the pro-
grammer can write expressions like \(-7\) or \(+5\). Cast operators associate from right to left and have the same precedence as other unary operators, such as unary + and unary -. This precedence is one level higher than that of the multiplicative operators *, / and % and one level lower than that of parentheses. (See the operator precedence chart in Appendix C.) We indicate the cast operator with the notation \((\text{type})\) in our precedence charts, to indicate that any type name can be used to form a cast operator.

**Common Programming Error 4.10**

Using floating-point numbers in a manner that assumes they are represented precisely can lead to incorrect results. Floating-point numbers are represented approximately by computers.

**Common Programming Error 4.11**

Assuming that integer division rounds (rather than truncates) can lead to incorrect results.

**Good Programming Practice 4.11**

Do not compare floating-point values for equality or inequality. Rather, test for whether the absolute value of the difference between two floating-point numbers is less than a specified small value.

Despite the fact that floating-point numbers are not always 100\% precise, they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is fine for most applications.

Another way in which floating-point numbers develop is through division. When we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

**4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)**

Let us work through another complete problem. We once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and we develop a corresponding Java program. Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the message “Enter result” on the screen each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results indicating the number of students who passed and the number of students who failed.
4. If more than 8 students passed the exam, print the message “Raise tuition.”
After reading the problem statement carefully, we make the following observations about the problem:

1. The program must process test results for 10 students. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it is a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
4. After the program has processed all the results, it must decide if more than eight students passed the exam.

Let us proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

*Analyze exam results and decide if tuition should be raised*

Once again, it is important to emphasize that the top is a complete representation of the program, but several refinements are likely before the pseudocode can evolve naturally into a Java program. Our first refinement is

*Initialize variables*

*Input the ten exam grades and count passes and failures*

*Print a summary of the exam results and decide if tuition should be raised*

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. We need counters to record the passes and failures, a counter to control the looping process and a variable to store the user input. The pseudocode statement

*Initialize variables*

may be refined as follows:

*Initialize passes to zero*

*Initialize failures to zero*

*Initialize student to one*

Notice that only the counters for the number of passes, number of failures and number of students are initialized. The pseudocode statement

*Input the ten quiz grades and count passes and failures*

requires a loop that successively inputs the result of each exam. Here it is known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., *nested* within the loop) a double-selection structure determines whether each exam result is a pass or a failure and increments the appropriate counter accordingly. The refinement of the preceding pseudocode statement is:
While student counter is less than or equal to ten
   Input the next exam result
   
   If the student passed
      Add one to passes
   else
      Add one to failures
   
   Add one to student counter

Notice the use of blank lines to set off the if/else control structure to improve program readability. The pseudocode statement

   Print a summary of the exam results and decide if tuition should be raised

may be refined as follows:

   Print the number of passes
   Print the number of failures

   If more than eight students passed
      Print “Raise tuition”

The complete second refinement appears in Fig. 4.10. Notice that the pseudocode also uses blank lines to set off the while structure for program readability.

---

Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
   Input the next exam result
   
   If the student passed
      Add one to passes
   else
      Add one to failures
   
   Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
   Print “Raise tuition”

---

Fig. 4.10 Pseudocode for examination-results problem.
This pseudocode is now sufficiently refined for conversion to Java. The Java program and two sample executions are shown in Fig. 4.11.

```java
// Fig. 4.11: Analysis.java
// Analysis of examination results.

import javax.swing.JOptionPane;

public class Analysis {

    // main method begins execution of Java application
    public static void main( String args[] )
    {
        // initializing variables in declarations
        int passes = 0,             // number of passes
        failures = 0,           // number of failures
        student = 1,            // student counter
        result;                 // one exam result
        String input,               // user-entered value
        output;              // output string

        // process 10 students; counter-controlled loop
        while ( student <= 10 ) {

            // obtain result from user
            input = JOptionPane.showInputDialog( 
                "Enter result (1=pass,2=fail)" );

            // convert result to int
            result = Integer.parseInt( input );

            // process result
            if ( result == 1 )
                passes = passes + 1;
            else
                failures = failures + 1;

            student = student + 1;
        }

        // termination phase
        output = "Passed: " + passes +
                 "\nFailed: " + failures;

        if ( passes > 8 )
            output = output + "\nRaise Tuition";

        JOptionPane.showMessageDialog( null, output, 
            "Analysis of Examination Results",
            JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );  // terminate application
    }
}
```

Fig. 4.11 Java program for examination-results problem (part 1 of 2).
Lines 13–18,

```java
int passes = 0, failures = 0, student = 1, result;
String input, output;
```

Fig. 4.11 Java program for examination-results problem (part 2 of 2).
declare the variables used in main to process the examination results. Note that we have taken advantage of a feature of Java that incorporates variable initialization into declarations (passes is assigned 0, failures is assigned 0 and student is assigned 1). Looping programs may require initialization at the beginning of each repetition; such initialization would normally occur in assignment statements.

Notice the nested if/else structure at lines 31–34 of the while structure’s body. Also, notice the use of String reference output in lines 40, 41 and 44 to build the string that lines 46–48 display in a message dialog.

**Good Programming Practice 4.12**

Initializing local variables when they are declared in methods helps the programmer avoid compiler messages warning of uninitialized data.

**Software Engineering Observation 4.8**

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working Java program from the algorithm is normally straightforward.

**Software Engineering Observation 4.9**

Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this method may work for simple and familiar problems, it can lead to serious errors in large, complex projects.

### 4.11 Assignment Operators

Java provides several assignment operators for abbreviating assignment expressions. For example, you can abbreviate the statement

\[
c = c + 3;
\]

with the addition assignment operator, `+=`, as

\[
c += 3;
\]

The `+=` operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

\[
variable = variable \text{ operator} \ expression;
\]

where `operator` is one of the binary operators `+`, `-`, `*`, `/` or `%` (or others we discuss later in the text), can be written in the form

\[
variable \ operator= \ expression;
\]

Thus, the assignment expression `c += 3` adds 3 to c. Figure 4.12 shows the arithmetic assignment operators, sample expressions using the operators and explanations of what the operators do.
Performance Tip 4.1

Programmers can write programs a bit faster and compilers can compile programs a bit faster when the abbreviated assignment operators are used. Some compilers generate code that runs faster when abbreviated assignment operators are used.

Performance Tip 4.2

Many of the performance tips we mention in this text result in nominal improvements, so the reader may be tempted to ignore them. Significant performance improvement is often realized when a supposedly nominal improvement is placed in a loop that may repeat a large number of times.

4.12 Increment and Decrement Operators

Java provides the unary increment operator, 

[math]\textbf{++} \quad \text{preincrement} \quad \textbf{a} \quad \text{Increment \textbf{a} by 1, then use the new value of \textbf{a} in the expression in which \textbf{a} resides.} \]

[math]\textbf{++} \quad \text{postincrement} \quad \textbf{a}++ \quad \text{Use the current value of \textbf{a} in the expression in which \textbf{a} resides, then increment \textbf{a} by 1.} \]

[math]\textbf{--} \quad \text{predecrement} \quad \textbf{b}-- \quad \text{Decrement \textbf{b} by 1, then use the new value of \textbf{b} in the expression in which \textbf{b} resides.} \]

[math]\textbf{--} \quad \text{postdecrement} \quad \textbf{b}-- \quad \text{Use the current value of \textbf{b} in the expression in which \textbf{b} resides, then decrement \textbf{b} by 1.} \]
Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, and then the new value of the variable is used in the expression in which it appears. Postincrementing (postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, and then the variable value is incremented (decremented) by 1.

The application in Fig. 4.14 demonstrates the difference between the preincrementing version and the postincrementing version of the ++ increment operator. Postincrementing the variable c causes it to be incremented after it is used in the System.out.println method call (line 13). Preincrementing the variable c causes it to be incremented before it is used in the System.out.println method call (line 20).

The program displays the value of c before and after the ++ operator is used. The decrement operator (--) works similarly.

**Good Programming Practice 4.13**

Unary operators should be placed next to their operands, with no intervening spaces.

```java
// Fig. 4.14: Increment.java
// Preincrementing and postincrementing
public class Increment {
    // main method begins execution of Java application
    public static void main( String args[] )
    {
        int c;

        c = 5;
        System.out.println( c ); // print 5
        System.out.println( c++ ); // print 5 then postincrement
        System.out.println( c ); // print 6

        System.out.println(); // skip a line

        c = 5;
        System.out.println( c ); // print 5
        System.out.println( ++c ); // preincrement then print 6
        System.out.println( c ); // print 6
    }
} // end main method
} // end class Increment
```

Fig. 4.14 The difference between preincrementing and postincrementing.
Line 16,

```java
System.out.println();    // skip a line
```
uses `System.out.println` to output a blank line. If `println` receives no arguments, it simply outputs a newline character.

The arithmetic assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 4.11 (lines 32, 34 and 36),

```java
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with assignment operators as

```java
passes += 1;
failures += 1;
student += 1;
```
with preincrement operators as

```java
++passes;
++failures;
++student;
```

or with postincrement operators as

```java
passes++;
failures++;
student++;
```

It is important to note here that when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect. It is only when a variable appears in the context of a larger expression that preincrementing and post-incrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

**Common Programming Error 4.12**

*Attempting to use the increment or decrement operator on an expression other than an lvalue is a syntax error. An lvalue is a variable or expression that can appear on the left side of an assignment operation. For example, writing `++(x + 1)` is a syntax error, because `(x + 1)` is not an lvalue.*

The chart in Fig. 4.15 shows the precedence and associativity of the operators that have been introduced up to this point. The operators are shown from top to bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. Notice that the conditional operator (`?:`), the unary operators increment (`++`), decrement (`--`), plus (`+`), minus (`-`) and casts and the assignment operators `=, +=, -=, *=, /=` and `%=` associate from right to left. All other operators in the operator precedence chart in Fig. 4.15 associate from left to right. The third column names the groups of operators.
4.13 Primitive Data Types

The table in Fig. 4.16 lists the primitive data types in Java. The primitive types are the
building blocks for more complicated types. Like its predecessor languages C and C++,
Java requires all variables to have a type before they can be used in a program. For this rea-
son, Java is referred to as a strongly typed language.

In C and C++ programs, programmers frequently had to write separate versions of pro-
grams to support different computer platforms, because the primitive data types were not
guaranteed to be identical from computer to computer. For example, an int value on one
machine might be represented by 16 bits (2 bytes) of memory, while an int value on
another machine might be represented by 32 bits (4 bytes) of memory. In Java, int values
are always 32 bits (4 bytes).

Portability Tip 4.1

Unlike in the programming languages C and C++, the primitive types in Java are portable
across all computer platforms that support Java. This and many other portability features of
Java enable programmers to write programs once, without knowing which computer plat-
form will execute the program. This attribute is sometimes referred to as WORA (Write Once
Run Anywhere).

Each data type in Fig. 4.16 is listed with its size in bits (there are eight bits to a byte)
and its range of values. Because the designers of Java want it to be maximally portable, they
chose to use internationally recognized standards for both character formats (Unicode) and
floating-point numbers (IEEE 754).

When instance variables of the primitive data types are declared in a class, they are
automatically assigned default values unless specified otherwise by the programmer.
Instance variables of types char, byte, short, int, long, float and double are
all given the value 0 by default. Variables of type boolean are given the value false
by default.
In “Thinking About Objects,” Section 3.8, we began the first phase of an object-oriented design (OOD) for our elevator simulator—identifying the classes needed to implement the simulator. We began by listing the nouns in the problem statement and then created a separate class for each category of noun and noun phrase that perform an important duty in the elevator simulation. We then represented the classes and their relationships in a UML class diagram (Fig. 3.23). Classes have attributes (data) and operations (behaviors). Class attributes are implemented in Java programs as variables; class behaviors are implemented as methods. In this section, we determine many of the class attributes needed to implement the elevator simulator. In Chapter 5, we examine how these attributes represent an object’s state, or condition. In Chapter 6, we determine class behavior. In Chapter 7, we concentrate on the interactions, often called collaborations, between the objects in the elevator simulator.
Consider the attributes of some real-world objects: A person’s attributes include height and weight, for example. A radio’s attributes include its station setting, its volume setting and whether it is set to AM or FM. A car’s attributes include its speedometer and odometer readings, the amount of gas in its tank, what gear it is in, etc. A personal computer’s attributes include its manufacturer (e.g., Sun, Apple, IBM or Compaq), type of screen (e.g., monochrome or color), main memory size (in megabytes), hard disk size (in gigabytes), etc.

We can identify the attributes of the classes in our system by looking for descriptive words and phrases in the problem statement. For each descriptive word or phrase we find, we create an attribute and assign that attribute to a class. We also create attributes to represent any additional data that a class may need (as the need for this data becomes clear throughout the design process).

We begin examining the problem statement looking for attributes distinct to each class. Figure 4.17 lists the words or phrases from the problem statement that describe each class. The sentence “The user can create any number of people in the simulation” implies that the model will introduce several Person objects during execution. We require an integer value representing the number of people in the simulation at any given time, because we may wish to track, or identify, the people in our model. As mentioned in Section 2.9, the ElevatorModel object acts as the “representative” for the model (even though the model consists of several classes) for interactions with other parts of the system (in this case, the user is a part of the system), so we assign the numberOfPeople attribute to class ElevatorModel.

Class Elevator contains several attributes. The phrases “is moving” and “is summoned” describe possible states of Elevator (we introduce states in the next “Thinking About Objects” section), so we include moving and summoned as boolean attributes. Elevator also arrives at a “destination floor,” so we include the attribute destinationFloor, representing the Floor at which the Elevator will arrive. Although the problem statement does not mention explicitly that the Elevator leaves from a current Floor, we may assume another attribute called currentFloor representing on which Floor the Elevator is resting. The problem statement specifies that “both the elevator and each floor have capacity for only one person,” so we include the capacity attribute for class Elevator (and class Floor) and set the value to 1. Lastly, the problem statement specifies that the elevator “takes five seconds to travel between floors,” so we introduce the travelTime attribute and set the value to 5.

Class Person contains several attributes. The user must be able to “create a unique person,” which implies that each Person object should have a unique identifier. We assign integer attribute ID to the Person object. The ID attribute helps to identify that Person object. In addition, the problem statement specifies that the Person can be “waiting on that floor to enter the elevator.” Therefore, “waiting” is a state that Person object may enter. Though not mentioned explicitly, if the Person is not waiting for the Elevator, the Person is moving to (or away from) the Elevator. We assign the boolean attribute moving to class Person. When this attribute is set to false, the Person is “waiting.” Lastly, the phrase “on that floor” implies that the Person occupies a floor. We cannot assign a Floor reference to class Person, because we are interested only in attributes. However, we want to include the location of the Person object in the model, so we include the currentFloor attribute, which may have a value of either 1 or 2.
Class **Floor** has a **capacity** attribute. The problem statement specified that the user could situate the person on either “the first or second floor”—therefore, a **Floor** object requires a value that distinguishes that **Floor** object as the first or second floor, so we include the **floorNumber** attribute.

According to the problem statement, the **ElevatorButton** and **FloorButton** are “pressed” by a **Person**. The buttons may be “reset” as well. The state of each button is either “pressed” or “reset.” We include the **boolean** attribute **pressed** in both button classes. When **pressed** is **true**, the button object is pressed; when **pressed** is **false**, the button object is reset. Classes **ElevatorDoor** and **FloorDoor** exhibit similar characteristics. Both objects are either “open” or “closed,” so we include the **boolean** attribute **open** in both door classes. Class **Light** also falls into this category—the light is either “illuminated” (turned on) or “turned off,” so we include the **boolean** attribute **on** in class **Light**. Note that although the problem statement mentions that the bell rings, there is no mention of when the bell “is ringing,” so we do not include a separate **ring** attribute for class **Bell**. As we progress through this case study, we will continue to add, modify and delete information about the classes in our system.

<table>
<thead>
<tr>
<th>Class</th>
<th>Descriptive words and phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElevatorModel</td>
<td>number of people in the simulation</td>
</tr>
<tr>
<td>ElevatorShaft</td>
<td>[no descriptive words or phrases]</td>
</tr>
</tbody>
</table>
| Elevator             | moving
                     summoned
                     current floor
                     destination floor
                     capacity of only one person
                     five seconds to travel between floors                          |
| Person               | unique
                     waiting / moving
                     current floor                                                  |
| Floor                | first or second; capacity for only one person                      |
| FloorButton          | pressed / reset                                                    |
| ElevatorButton       | pressed / reset                                                    |
| FloorDoor            | door closed / door open                                            |
| ElevatorDoor         | door closed / door open                                            |
| Bell                 | [no descriptive words or phrases]                                  |
| Light                | illuminated / turned off                                          |

Fig. 4.17  Descriptive words and phrases from problem statement.
Figure 4.18 is a class diagram that lists some of the attributes for each class in our system—the descriptive words and phrases in Fig. 4.17 help us generate these attributes. Note that Fig. 4.18 does not show associations among objects—we showed these associations in Fig. 3.23. In the UML class diagram, a class’s attributes are placed in the middle compartment of the class’s rectangle. Consider the open attribute of class ElevatorDoor:

\[
\text{open : Boolean} = \text{false}
\]

This listing contains three pieces of information about the attribute. The attribute name is open. The attribute type is Boolean. The type depends on the language used to write the software system. In Java, for example, the value can be a primitive type, such as boolean or float, as well as a user-defined type like a class—we begin our study of classes in Chapter 8, where we will see that each new class is a new data type.

We can also indicate an initial value for each attribute. The open attribute in class ElevatorDoor has an initial value of false. If a particular attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the ID attribute of class Person is an integer—in Java, the ID attribute is of type int. Here we show no initial value, because the value of this attribute is a number that we do not yet know; this number will be determined by ElevatorModel at execution time. Integer attribute currentFloor for class Person is not determined until program execution as well—this attribute is determined when the simulation user decides on which Floor to place the Person. For now we do not concern ourselves with the types or initial values of the attributes. We include only the information we can glean easily from the problem statement.

Note that Fig. 4.18 does not include attributes for class ElevatorShaft. Actually, class ElevatorShaft contains seven attributes that we can determine from the class diagram of Fig. 3.23—references to the Elevator object, two FloorButton objects, two FloorDoor objects and two Light objects. Class ElevatorModel contains three user-defined attributes—two references to Floor objects and a reference to the ElevatorShaft object. Class Elevator also contains three user-defined attributes—reference to the ElevatorButton object, the ElevatorDoor object and the Bell object. To save space, we will not show these additional attributes in our class diagrams—we will, however, include them in the code in the appendices.

The class diagram of Fig. 4.18 provides a good basis for the structure of our model but the diagram is not fully complete. For example, the attribute currentFloor in class Person represents the floor on which a person is currently located. However, on what floor is the person when that person rides the elevator? These attributes do not yet sufficiently represent the structure of the model. As we present more of the UML and object-oriented design through Chapter 22, we will continue to strengthen the structure of our model.

---

2. Note that the attribute types in Fig. 4.18 are in UML notation. We will associate the attribute types Boolean and Integer in the UML diagram with the attribute types boolean and int in Java, respectively. We described in Chapter 3 that Java provides a “type-wrapper class” for each primitive data type. The Java type-wrapper classes have the same notation as the UML notation for attribute types; however, when we implement our design in Java starting in Chapter 8, we use primitive data types for simplification. Deciding whether to use primitive data types or type-wraper classes is an implementation-specific issue that should not be mentioned in the UML.
SUMMARY

- A procedure for solving a problem in terms of the actions to be executed and the order in which the actions should be executed is called an algorithm.
- Specifying the order in which statements execute in a computer program is called program control.
- Pseudocode helps the programmer “think out” a program before attempting to write it in a programming language, such as Java.
- Top-down, stepwise refinement is a process for refining pseudocode by maintaining a complete representation of the program during each refinement.
- Declarations are messages to the compiler telling it the names and attributes of variables and telling it to reserve space for variables.
- A selection structure chooses among alternative courses of action.
- The if selection structure executes an indicated action only when the condition is true.
- The if/else selection structure specifies separate actions to execute when the condition is true and when the condition is false.
- When more than one statement should execute where normally only a single statement appears, the statements must be enclosed in braces, forming a block. A block can be placed anywhere a single statement can be placed.
- An empty statement, indicating that no action is to be taken, is indicated by placing a semicolon (;) where a statement would normally be.
- A repetition structure specifies that an action is to be repeated while some condition remains true.
- The format for the while repetition structure is
  
  \[
  \text{while ( condition )} \\
  \text{statement}
  \]
• A value that contains a fractional part is referred to as a floating-point number and is represented by the data type `float` or `double`.

• Unary cast operator `(double)` creates a temporary floating-point copy of its operand.

• Java provides the arithmetic assignment operators `+=, -=, *=, /=` and `%=`, which help abbreviate certain common types of expressions.

• The increment operator, `++`, and the decrement operator, `--`, increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1 first, and then used in its expression. If the operator is postfixed to the variable, the variable is used in its expression, and then incremented or decremented by 1.

• The primitive types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`) are the building blocks for more complicated types in Java.

• Java requires all variables to have a type before they can be used in a program. For this reason, Java is referred to as a strongly typed language.

• Primitive types in Java are portable across all computer platforms that support Java.

• Java uses internationally recognized standards for both character formats (Unicode) and floating-point numbers (IEEE 754).

• Instance variables of types `char`, `byte`, `short`, `int`, `long`, `float` and `double` are all given the value 0 by default. Variables of type `boolean` are given the value false by default.

**TERMINOLOGY**

- `--` operator
- `?:` operator
- `++` operator
- action
- action/decision model
- algorithm
- arithmetic assignment operators: `+=, -=, *=, /=` and `%=`
- block
- body of a loop
- cast operator, `(type)`
- conditional operator `(?:)`
- control structure
- counter-controlled repetition
- decision
- decrement operator `(--)`
- definite repetition
- `double`
- double-selection structure
- empty statement `;`
- `if` selection structure
- `if/else` selection structure
- implicit conversion
- increment operator `[++`)
- indefinite repetition
- infinite loop
- initialization
- integer division
- ISO Unicode character set
- logic error
- loop counter
- loop-continuation condition
- nested control structures
- postdecrement operator
- postincrement operator
- predecrement operator
- preincrement operator
- promotion
- pseudocode
- repetition
- repetition structures
- selection
- sentinel value
- sequential execution
- single-entry/single-exit control structures
- single-selection structure
- stacked control structures
- structured programming
- syntax error
- top-down, stepwise refinement
- unary operator
- `while` repetition structure
- white-space characters
SELF-REVIEW EXERCISES

4.1 Fill in the blanks in each of the following statements:
   a) All programs can be written in terms of three types of control structures: ________, ________, and ________.
   b) The ________ selection structure is used to execute one action when a condition is true and another action when that condition is false.
   c) Repeating a set of instructions a specific number of times is called ________ repetition.
   d) When it is not known in advance how many times a set of statements will be repeated, a ________ value can be used to terminate the repetition.

4.2 Write four different Java statements that each add 1 to integer variable \( x \).

4.3 Write Java statements to accomplish each of the following tasks:
   a) Assign the sum of \( x \) and \( y \) to \( z \), and increment the value of \( x \) by 1 after the calculation. Use only one statement.
   b) Test if the value of the variable \( \text{count} \) is greater than 10. If it is, print "Count is greater than 10".
   c) Decrement the variable \( x \) by 1, and then subtract it from the variable \( \text{total} \). Use only one statement.
   d) Calculate the remainder after \( q \) is divided by \( \text{divisor} \), and assign the result to \( q \). Write this statement in two different ways.

4.4 Write a Java statement to accomplish each of the following tasks:
   a) Declare variables \( \text{sum} \) and \( x \) to be of type \( \text{int} \).
   b) Assign 1 to variable \( x \).
   c) Assign 0 to variable \( \text{sum} \).
   d) Add variable \( x \) to variable \( \text{sum} \), and assign the result to variable \( \text{sum} \).
   e) Print "The sum is: \( \)", followed by the value of variable \( \text{sum} \).

4.5 Combine the statements that you wrote in Exercise 4.4 into a Java application that calculates and prints the sum of the integers from 1 to 10. Use the \( \text{while} \) structure to loop through the calculation and increment statements. The loop should terminate when the value of \( x \) becomes 11.

4.6 Determine the value of each variable after the calculation is performed. Assume that when each statement begins executing, all variables have the integer value 5.
   a) \( \text{product} *= x++ \);
   b) \( \text{quotient} /= ++x \);

4.7 Identify and correct the errors in each of the following sets of code:
   a) \( \text{while} ( \ c <= 5 \ ) \ { \)
      \( \text{product} *= c; \)
      \( ++c; \)
   b) \( \text{if} ( \ \text{gender} == 1 \ ) \)
      \( \text{System.out.println} ( \ "\text{Woman}" \ ); \)
      \( \text{else}; \)
      \( \text{System.out.println} ( \ "\text{Man}" \ ); \)

4.8 What is wrong with the following \( \text{while} \) repetition structure?
   \( \text{while} ( \ z >= 0 \ ) \)
   \( \text{sum} += z; \)

ANSWERS TO SELF-REVIEW EXERCISES

4.1 a) sequence, selection, repetition. b) \( \text{if/else} \). c) counter-controlled, or definite. d) Sentinel, signal, flag or dummy.
4.2  \[ \begin{align*} 
&x = x + 1; \\
&x += 1; \\
&++x; \\
&x++; 
\end{align*} \]

4.3  \[ \begin{align*} 
&a) \quad z &= x++ + y; \\
&b) \quad \text{if ( count > 10 )} & \quad \text{System.out.println( "Count is greater than 10" );} \\
&c) \quad \text{total} &= --x; \\
&d) \quad q &= \text{divisor}; & \quad q = q \% \text{ divisor;} 
\end{align*} \]

4.4  \[ \begin{align*} 
&a) \quad \text{int} \quad \text{sum, x;} \\
&b) \quad x &= 1; \\
&c) \quad \text{sum} &= 0; \\
&d) \quad \text{sum} &= x; \text{ or sum} &= \text{sum} + x; \\
&e) \quad \text{System.out.println( "The sum is: " + sum );} 
\end{align*} \]

4.5  The program is as follows:

```java
// Calculate the sum of the integers from 1 to 10
public class Calculate {
    public static void main( String args[] )
    {
        int sum, x;
        x = 1;
        sum = 0;
        while ( x <= 10 ) {
            sum += x;
            ++x;
        }
        System.out.println( "The sum is: " + sum );
    }
}
```

4.6  a) \textbf{product} = 25, \textbf{x} = 6 \\
    b) \textbf{quotient} = 0, \textbf{x} = 6

4.7  a) Error: Missing the closing right brace of the while structure’s body. 
    Correction: Add a closing right brace after the statement ++c;.
    b) Error: Semicolon after else results in a logic error. The second output statement will always be executed. 
    Correction: Remove the semicolon after else.

4.8  The value of the variable \textbf{z} is never changed in the while structure. Therefore, if the loop-continuation condition \((\textbf{z} \geq 0)\) is true, an infinite loop is created. To prevent an infinite loop from occurring, \textbf{z} must be decremented so that it eventually becomes less than 0.

**EXERCISES**

4.9  Identify and correct the errors in each of the following sets of code. \textit{Note: There may be more than one error in each piece of code}:
a) if ( age >= 65 );
    System.out.println("Age greater than or equal to 65");
else
    System.out.println("Age is less than 65");
b) int x = 1, total;
    while ( x <= 10 ) {
        total += x;
        ++x;
    }
c) While ( x <= 100 )
    total += x;
    ++x;
d) while ( y > 0 ) {
    System.out.println( y );
    ++y;

4.10 What does the following program print?

```java
public class Mystery {
    public static void main(String args[]) {
        int y, x = 1, total = 0;
        while ( x <= 10 ) {
            y = x * x;
            System.out.println( y );
            total += y;
            ++x;
        }
        System.out.println("Total is "+ total);
    }
}
```

For Exercise 4.11 through Exercise 4.14, perform each of the following steps:
   a) Read the problem statement.
   b) Formulate the algorithm using pseudocode and top-down, stepwise refinement.
   c) Write a Java program.
   d) Test, debug and execute the Java program.
   e) Process three complete sets of data.

4.11 Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a Java application that will input the miles driven and gallons used (both as integers) for each tankful. The program should calculate and display the miles per gallon obtained for each tankful and print the combined miles per gallon obtained for all tankfuls up to this point. All averaging calculations should produce floating-point results. Use input dialogs to obtain the data from the user.

4.12 Develop a Java application that will determine if a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:
   a) account number,
   b) balance at the beginning of the month,
c) total of all items charged by the customer this month,
d) total of all credits applied to the customer's account this month, and
e) allowed credit limit.

The program should input each of these facts from input dialogs as integers, calculate the new balance \(= \text{beginning balance} + \text{charges} - \text{credits}\), display the new balance and determine if the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the message “Credit limit exceeded.”

4.13 A large company pays its salespeople on a commission basis. The salespeople receive $200 per week, plus 9% of their gross sales for that week. For example, a salesperson who sells $5000 worth of merchandise in a week receives $200 plus 9% of $5000, or a total of $650. You have been supplied with a list of items sold by each salesperson. The values of these items are as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>239.99</td>
</tr>
<tr>
<td>2</td>
<td>129.75</td>
</tr>
<tr>
<td>3</td>
<td>99.95</td>
</tr>
<tr>
<td>4</td>
<td>350.89</td>
</tr>
</tbody>
</table>

Develop a Java application that inputs one salesperson's items sold for last week and calculates and displays that salesperson's earnings. There is no limit to the number of items that can be sold by a salesperson.

4.14 Develop a Java application that will determine the gross pay for each of three employees. The company pays “straight time” for the first 40 hours worked by each employee and pays “time and a half” for all hours worked in excess of 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee's gross pay. Use input dialogs to input the data.

4.15 The process of finding the largest value (i.e., the maximum of a group of values) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program and then a Java application that inputs a series of 10 single-digit numbers as characters and determines and prints the largest of the numbers. **Hint:** Your program should use the following three variables:

a) **counter**: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed);
b) **number**: The current digit input to the program;
c) **largest**: The largest number found so far.

4.16 Write a Java application that uses looping to print the following table of values:

<table>
<thead>
<tr>
<th>N</th>
<th>10*N</th>
<th>100*N</th>
<th>1000*N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>200</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>300</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>400</td>
<td>4000</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>500</td>
<td>5000</td>
</tr>
</tbody>
</table>

4.17 Using an approach similar to that for Exercise 4.15, find the two largest values of the 10 digits entered. [**Note:** You may input each number only once.]
4.18 Modify the program in Fig. 4.11 to validate its inputs. For any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

4.19 What does the following program print?

```java
public class Mystery2 {
    public static void main( String args[] )
    {
        int count = 1;
        while ( count <= 10 ) {
            System.out.println(count % 2 == 1 ? "*****" : "########");
            ++count;
        }
    }
}
```

4.20 What does the following program print?

```java
public class Mystery3 {
    public static void main( String args[] )
    {
        int row = 10, column;
        while ( row >= 1 ) {
            column = 1;
            while ( column <= 10 ) {
                System.out.print(row % 2 == 1 ? "<" : ">" );
                ++column;
            }
            --row;
            System.out.println();
        }
    }
}
```

4.21 (Dangling-Else Problem) Determine the output for each of the given sets of code when \( x \) is 9 and \( y \) is 11 and when \( x \) is 11 and \( y \) is 9. Note that the compiler ignores the indentation in a Java program. Also, the Java compiler always associates an \texttt{else} with the previous \texttt{if} unless told to do otherwise by the placement of braces ({}). On first glance, the programmer may not be sure which \texttt{if} an \texttt{else} matches; this situation is referred to as the “dangling-else problem.” We have eliminated the indentation from the following code to make the problem more challenging. [Hint: Apply indentation conventions you have learned.]

a) \texttt{if ( x < 10 )}
   \texttt{if ( y > 10 )}
   \texttt{System.out.println( "*****" );}
   \texttt{else}
   \texttt{System.out.println( "#####" );}
   \texttt{System.out.println( "$$$$$$" );}
b) \[
\textbf{if} \ ( \ x < 10 \ ) \ \{ \\
\quad \textbf{if} \ ( \ y > 10 \ ) \\
\quad \quad \text{System.out.println("*****");} \\
\quad \text{else} \ \\
\quad \quad \text{System.out.println("####")}; \\
\quad \quad \text{System.out.println("$$$$");} \\
\quad \}
\]

4.22 (Another Dangling-Else Problem) Modify the given code to produce the output shown in each part of the problem. Use proper indentation techniques. You may not make any changes other than inserting braces and changing the indentation of the code. The compiler ignores indentation in a Java program. We have eliminated the indentation from the given code to make the problem more challenging. [Note: It is possible that no modification is necessary for some of the parts.]

\[
\textbf{if} \ ( \ y == 8 \ ) \\
\textbf{if} \ ( \ x == 5 \ ) \\
\quad \text{System.out.println("@@@" }); \\
\quad \text{else} \\
\quad \quad \text{System.out.println("####")}; \\
\quad \quad \text{System.out.println("$$$$");} \\
\quad \quad \text{System.out.println("&&&&" );}
\]

a) Assuming that \(x = 5\) and \(y = 8\), the following output is produced:

```
@@@@@
$$$$$
&&&&&
```

b) Assuming that \(x = 5\) and \(y = 8\), the following output is produced:

```
@@@@@
```

c) Assuming that \(x = 5\) and \(y = 8\), the following output is produced:

```
@@@@@
&&&&
```

d) Assuming that \(x = 5\) and \(y = 7\), the following output is produced [Note: The last three output statements after the \textbf{else} are all part of a block]:

```
#####
$$$$$
&&&&
```

4.23 Write an applet that reads in the size of the side of a square and displays a hollow square of that size out of asterisks, by using the \texttt{drawString} method inside your applet’s \texttt{paint} method.
Use an input dialog to read the size from the user. Your program should work for squares of all lengths of side between 1 and 20.

4.24 A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers are palindromes: 12321, 55555, 45554 and 11611. Write an application that reads in a five-digit integer and determines whether or not it is a palindrome. If the number is not five digits long, display an error message dialog indicating the problem to the user. When the user dismisses the error dialog, allow the user to enter a new value.

4.25 Write an application that inputs an integer containing only 0s and 1s (i.e., a “binary” integer) and prints its decimal equivalent. [Hint: Use the modulus and division operators to pick off the “binary number’s” digits one at a time, from right to left. Just as in the decimal number system, where the rightmost digit has a positional value of 1 and the next digit to the left has a positional value of 10, then 100, then 1000, etc., in the binary number system the rightmost digit has a positional value of 1, the next digit to the left has a positional value of 2, then 4, then 8, etc. Thus, the decimal number 234 can be interpreted as $4 \times 1 + 3 \times 10 + 2 \times 100$. The decimal equivalent of binary 1101 is $1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8$, or $1 + 0 + 4 + 8$ or, 13.]

4.26 Write an application that displays the following checkerboard pattern:

```
********
********
********
********
********
********
********
********
```

Your program may use only three output statements, one of the form

```java
System.out.print( "* " );
```

one of the form

```java
System.out.print( " " );
```

and one of the form

```java
System.out.println();
```

Note that the preceding statement indicates that the program should output a single newline character to drop to the next line of the output. [Hint: Repetition structures are required in this exercise.]

4.27 Write an application that keeps displaying in the command window the multiples of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?

4.28 What is wrong with the following statement? Provide the correct statement to add one to the sum of x and y.

```java
System.out.println( ++(x + y) );
```

4.29 Write an application that reads three nonzero values entered by the user in input dialogs and determines and prints if they could represent the sides of a triangle.
4.30 Write an application that reads three nonzero integers and determines and prints if they could represent the sides of a right triangle.

4.31 A company wants to transmit data over the telephone, but it is concerned that its phones may be tapped. All of its data are transmitted as four-digit integers. It has asked you to write a program that will encrypt its data so that the data may be transmitted more securely. Your application should read a four-digit integer entered by the user in an input dialog and encrypt it as follows: Replace each digit by \((\text{the sum of that digit plus 7}) \mod 10\). Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it to form the original number.

4.32 The factorial of a nonnegative integer \(n\) is written as \(n!\) (pronounced “\(n\) factorial”) and is defined as follows:
\[
n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 \quad \text{for values of } n \text{ greater than or equal to } 1
\]
and
\[
n! = 1 \quad \text{for } n = 0.
\]
For example, \(5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1\), which is 120.

a) Write an application that reads a nonnegative integer from an input dialog and computes and prints its factorial.

b) Write an application that estimates the value of the mathematical constant \(e\) by using the formula
\[
e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots
\]

c) Write an application that computes the value of \(e^x\) by using the formula:
\[
e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots
\]
Objectives

• To be able to use the **for** and **do/while** repetition structures to execute statements in a program repeatedly.
• To understand multiple selection using the **switch** selection structure.
• To be able to use the **break** and **continue** program control statements.
• To be able to use the logical operators.

*Who can control his fate?*
William Shakespeare, *Othello*

*The used key is always bright.*
Benjamin Franklin

*Man is a tool-making animal.*
Benjamin Franklin

*Intelligence … is the faculty of making artificial objects, especially tools to make tools.*
Henri Bergson
5.1 Introduction

Chapter 4 began our introduction to the types of building blocks that are available for problem solving and used those building blocks to employ proven program construction principles. In this chapter, we continue our presentation of the theory and principles of structured programming by introducing Java’s remaining control structures. As in Chapter 4, the Java techniques you learn here are applicable to most high-level languages. When we begin our formal treatment of object-based programming in Java in Chapter 8, we will see that the control structures we study in this chapter and Chapter 4 are helpful in building and manipulating objects.

5.2 Essentials of Counter-Controlled Repetition

Counter-controlled repetition requires the following:

1. the name of a control variable (or loop counter),
2. the initial value of the control variable,
3. the amount of increment (or decrement) by which the control variable is modified each time through the loop (also known as each iteration of the loop), and
4. the condition that tests for the final value of the control variable (i.e., whether looping should continue).

To see the four elements of counter-controlled repetition, consider the applet shown in Fig. 5.1, which draws 10 lines from the applet’s paint method. Remember that an applet requires a separate HTML document to load the applet into the appletviewer or a browser. For the purpose of this applet, the <applet> tag specifies a width of 275 pixels and a height of 110 pixels.
The applet's `paint` method (that the applet container calls when it executes the applet) operates as follows: The declaration on line 18 names the control variable (`counter`), declares it to be an integer, reserves space for it in memory and sets it to an initial value of 1. Declarations that include initialization are, in effect, executable statements. The declaration and initialization of `counter` could also have been accomplished with the declaration and statement

```java
int counter = 1; // declare counter
```

The declaration is not executable, but the assignment statement is. We use both methods of initializing variables throughout this book.

```java
// Fig. 5.1: WhileCounter.java
// Counter-controlled repetition

// Java core packages
import java.awt.Graphics;

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

public class WhileCounter extends JApplet {

    // draw lines on applet's background
    public void paint( Graphics g ) {
        // call inherited version of method paint
        super.paint( g );

        int counter = 1; // initialization

        while ( counter <= 10 ) { // repetition condition
            g.drawLine( 10, 10, 250, counter * 10 );
            ++counter; // increment
        } // end while structure
    } // end method paint

} // end class WhileCounter
```

Fig. 5.1 Counter-controlled repetition.
Line 21 in the `while` structure uses `Graphics` reference `g`, which refers to the applet’s `Graphics` object, to send the `drawLine` message to the `Graphics` object, asking it to draw a line. Remember that “sending a message to an object” actually means calling a method to perform a task. One of the `Graphics` object’s many services is to draw lines. In previous chapters, we also saw that the `Graphics` object’s other services include drawing rectangles, strings and ovals. `Graphics` method `drawLine` requires four arguments, representing the line’s first x-coordinate, first y-coordinate, second x-coordinate and second y-coordinate. In this example, the second y-coordinate changes value during each iteration of the loop with the calculation `counter * 10`. This change causes the second point (the end point of the line) in each call to `drawLine` to move 10 pixels down the applet’s display area.

Line 22 in the `while` structure increments the control variable by 1 for each iteration of the loop. The loop-continuation condition in the `while` structure tests whether the value of the control variable is less than or equal to 10 (the final value for which the condition is `true`). Note that the program performs the body of this `while` structure even when the control variable is 10. The loop terminates when the control variable exceeds 10 (i.e., `counter` becomes 11).

The program in Fig. 5.1 can be made more concise by initializing `counter` to 0 and preincrementing `counter` in the `while` structure condition as follows:

```java
while ( ++counter <= 10 )     // repetition condition
g.drawLine( 10, 10, 250, counter * 10 );
```

This code saves a statement (and eliminates the need for braces around the loop’s body), because the `while` condition performs the increment before testing the condition. (Remember that the precedence of `++` is higher than that of `<=`.) Coding in such a condensed fashion takes practice.

**Good Programming Practice 5.1**

Programs should control counting loops with integer values.

**Common Programming Error 5.1**

Because floating-point values may be approximate, controlling the counting of loops with floating-point variables may result in imprecise counter values and inaccurate tests for termination.

**Good Programming Practice 5.2**

Indent the statements in the body of each control structure.

**Good Programming Practice 5.3**

Put a blank line before and after each major control structure to make it stand out in the program.

**Good Programming Practice 5.4**

Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of nesting.
Good Programming Practice 5.5

Vertical spacing above and below control structures, and indentation of the bodies of control structures within the control structures' headers, gives programs a two-dimensional appearance that enhances readability.

5.3 The for Repetition Structure

The for repetition structure handles all of the details of counter-controlled repetition. To illustrate the power of the for structure, let us rewrite the applet of Fig. 5.1. The result is shown in Fig. 5.2. Remember that this program requires a separate HTML document to load the applet into the appletviewer. For the purpose of this applet, the <applet> tag specifies a width of 275 pixels and a height of 110 pixels.

```java
// Fig. 5.2: ForCounter.java
// Counter-controlled repetition with the for structure
import java.awt.Graphics;
import javax.swing.JApplet;
public class ForCounter extends JApplet {
    // draw lines on applet's background
    public void paint( Graphics g )
    {
        // call inherited version of method paint
        super.paint( g );
        // Initialization, repetition condition and incrementing
        // are all included in the for structure header.
        for ( int counter = 1; counter <= 10; counter++ )
            g.drawLine( 10, 10, 250, counter * 10 );
    }
} // end class ForCounter
```

Fig. 5.2 Counter-controlled repetition with the for structure.
The applet’s `paint` method operates as follows: When the `for` structure (lines 20–21) begins executing, the control variable `counter` is initialized to 1. (The first two elements of counter-controlled repetition and the name of the control variable and its initial value.) Next, the program checks the loop-continuation condition, `counter <= 10`. The condition contains the final value (10) of the control variable. Because the initial value of `counter` is 1, the condition is satisfied (true), so the body statement (line 21) draws a line. After executing the body of the loop, the program increments variable `counter` in the expression `counter++`. Then, the program performs the loop-continuation test again to determine whether the program should continue with the next iteration of the loop or whether it should terminate the loop. At this point, the control variable value is 2, so the condition is true (i.e., the final value is not exceeded), and thus the program performs the body statement again (i.e., the next iteration of the loop). This process continues until the `counter`’s value becomes 11, causing the loop-continuation test to fail and repetition to terminate. Then, the program performs the first statement after the `for` structure. (In this case, method `paint` terminates, because the program reaches the end of `paint`.)

Notice that Fig. 5.2 uses the loop-continuation condition `counter <= 10`. If the programmer incorrectly specified `counter < 10` as the condition, the loop would be executed only nine times. This mistake is a common logic error called an off-by-one error.

**Common Programming Error 5.2**

Using an incorrect relational operator or using an incorrect final value of a loop counter in the condition of a `while`, `for` or `do/while` structure can cause an off-by-one error.

**Good Programming Practice 5.6**

Using the final value in the condition of a `while` or `for` structure and using the `<=` relational operator will help avoid off-by-one errors. For a loop that prints the values 1 to 10, the loop-continuation condition should be `counter <= 10` rather than `counter < 10` (which causes an off-by-one error) or `counter < 11` (which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times, `counter` would be initialized to zero and the loop-continuation test would be `counter < 10`.

Figure 5.3 takes a closer look at the `for` structure of Fig. 5.2. The `for` structure’s first line (including the keyword `for` and everything in parentheses after `for`) is sometimes called the `for` structure header. Notice that the `for` structure “does it all”: It specifies each of the items needed for counter-controlled repetition with a control variable. If there is more than one statement in the body of the `for` structures, braces (`{` and `}`) are required to define the body of the loop.

<table>
<thead>
<tr>
<th>for keyword</th>
<th>Control variable name</th>
<th>Final value of control variable</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for ( int counter = 1; counter &lt;= 10; counter++ )</code></td>
<td><code>initial value of control variable</code></td>
<td><code>loop-continuation condition</code></td>
</tr>
</tbody>
</table>
The general format of the for structure is

```
for ( expression1; expression2; expression3 )
      statement
```

where expression1 names the loop’s control variable and provides its initial value, expression2 is the loop-continuation condition (containing the control variable’s final value) and expression3 modifies the value of the control variable, so that the loop-continuation condition eventually becomes false. In most cases, the for structure can be represented with an equivalent while structure, with expression1, expression2 and expression3 placed as follows:

```
expression1;
while ( expression2 ) {
      statement
      expression3;
}
```

In Section 5.7, we show a case in which a for structure cannot be represented with an equivalent while structure.

If expression1 (the initialization section) declares the control variable inside the parentheses of the header of the for structure (i.e., the control variable’s type is specified before the name of the variable), the control variable can be used only in the for structure. This restricted use of the name of the control variable is known as the variable’s scope. The scope of a variable defines where the program can use the variable. For example, we mentioned previously that a program can use a local variable only in the method that declares the variable. Scope is discussed in detail in Chapter 6, “Methods.”

**Common Programming Error 5.3**

When the control variable of a for structure is initially defined in the initialization section of the header of the for structure, using the control variable after the body of the structure is a syntax error.

Sometimes, expression1 and expression3 in a for structure are comma-separated lists of expressions that enable the programmer to use multiple initialization expressions and/or multiple increment expressions. For example, there may be several control variables in a single for structure that must be initialized and incremented.

**Good Programming Practice 5.7**

Place only expressions involving the control variables in the initialization and increment sections of a for structure. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the body of the loop (if they execute once per iteration of the loop, like incrementing or decrementing statements).

The three expressions in the for structure are optional. If expression2 is omitted, Java assumes that the loop-continuation condition is true, thus creating an infinite loop. One might omit expression1 if the program initializes the control variable before the loop. One might omit expression3 if the program calculates the increment with statements in the loop’s body or if the loop does not require an increment. The increment expression in the for structure acts as a stand-alone statement at the end of the body of the for structure, so the expressions
counter = counter + 1
counter += 1
++counter
counter++

are equivalent in the increment portion of the for structure. Many programmers prefer the form counter++, because a for structure increments its control variable after the body of the loop executes and placing ++ after the variable name increments the variable after the program uses its value. Therefore, the postincrementing form seems more natural. Preincrementing and postincrementing have the same effect in the increment expression, because the increment does not appear in a larger expression. The two semicolons in the for structure are required.

Common Programming Error 5.4
Using commas instead of the two required semicolons in a for header is a syntax error.

Common Programming Error 5.5
Placing a semicolon immediately to the right of the right parenthesis of a for header makes the body of that for structure an empty statement. This is normally a logic error.

The initialization, loop-continuation condition and increment portions of a for structure can contain arithmetic expressions. For example, assume that \( x = 2 \) and \( y = 10 \). If \( x \) and \( y \) are not modified in the body of the loop, the statement

\[
\text{for ( int j = x; j <= 4 * x * y; j += y / x )}
\]

is equivalent to the statement

\[
\text{for ( int j = 2; j <= 80; j += 5 )}
\]

The increment of a for structure may also be negative, in which case it is really a decrement, and the loop actually counts downward.

If the loop-continuation condition is initially false, the program does not perform the body of the for structure. Instead, execution proceeds with the statement following the for structure.

Programs frequently display the control variable value or use it in calculations in loop body. However, this use is not required. It is common to use the control variable for controlling repetition while never mentioning it in the body of the for structure.

Testing and Debugging Tip 5.1
Although the value of the control variable can be changed in the body of a for loop, avoid doing so, because this practice can lead to subtle errors.

We flowchart the for structure much as we do the while structure. For example, the flowchart of the for statement

\[
\text{for ( int counter = 1; counter <= 10; counter++ )}
\]

\[
g\text{.drawLine( 10, 10, 250, counter * 10 );}
\]

is shown in Fig. 5.4. This flowchart makes it clear that the initialization occurs only once and that the increment occurs each time after the program performs the body statement.
Note that, besides small circles and arrows, the flowchart contains only rectangle symbols and a diamond symbol. The programmer fills the rectangles and diamonds with actions and decisions appropriate to the algorithm.

### 5.4 Examples Using the for Structure

The examples given next show methods of varying the control variable in a for structure. In each case, we write the appropriate for structure header. Note the change in the relational operator for loops that decrement the control variable.

- **a)** Vary the control variable from 1 to 100 in increments of 1.
  
  ```
  for ( int i = 1; i <= 100; i++ )
  ```

- **b)** Vary the control variable from 100 to 1 in increments of -1 (i.e., decrements of 1).
  
  ```
  for ( int i = 100; i >= 1; i-- )
  ```

- **c)** Vary the control variable from 7 to 77 in steps of 7.
  
  ```
  for ( int i = 7; i <= 77; i += 7 )
  ```

- **d)** Vary the control variable from 20 to 2 in steps of -2.
  
  ```
  for ( int i = 20; i >= 2; i -= 2 )
  ```

- **e)** Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17, 20.
  
  ```
  for ( int j = 2; j <= 20; j += 3 )
  ```

- **f)** Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.
  
  ```
  for ( int j = 99; j >= 0; j -= 11 )
  ```

---

**Fig. 5.4** Flowcharting a typical for repetition structure.
Common Programming Error 5.6

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (such as using \( i <= 1 \) in a loop counting down to 1) is usually a logic error and will yield incorrect results when the program runs.

The next two sample programs demonstrate simple applications of the \texttt{for} repetition structure. The application in Fig. 5.5 uses the \texttt{for} structure to sum all the even integers from 2 to 100. Remember that the \texttt{java} interpreter is used to execute an application from the command window.

Note that the body of the \texttt{for} structure in Fig. 5.5 could actually be merged into the rightmost portion of the \texttt{for} header by using a comma as follows:

\begin{verbatim}
for ( int number = 2;
    number <= 100;
    sum += number, number += 2 )
; // empty statement
\end{verbatim}

Fig. 5.5 Summation with the \texttt{for} structure.
Similarly, the initialization \texttt{sum = 0} could be merged into the initialization section of the \texttt{for} structure.

**Good Programming Practice 5.8**

Although statements preceding a \texttt{for} structure and statements in the body of a \texttt{for} structure can often be merged into the header of the \texttt{for} structure, avoid doing so, because it makes the program more difficult to read.

**Good Programming Practice 5.9**

Limit the size of control structure headers to a single line, if possible.

The next example uses the \texttt{for} structure to compute compound interest. Consider the following problem:

A person invests $1000.00 in a savings account yielding 5\% interest. Assuming that all interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

\[
a = p (1 + r)^n
\]

where

- \(p\) is the original amount invested (i.e., the principal)
- \(r\) is the annual interest rate
- \(n\) is the number of years
- \(a\) is the amount on deposit at the end of the \(n\)th year.

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. The solution is the application shown in Fig. 5.6.

```java
// Fig. 5.6: Interest.java
// Calculating compound interest

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

// Java extension packages
import javax.swing.JOptionPane;
import javax.swing.JTextArea;

public class Interest {

    // main method begins execution of Java application
    public static void main( String args[] )
    {
        double amount, principal = 1000.0, rate = 0.05;

        // create DecimalFormat to format floating-point numbers
        // with two digits to the right of the decimal point
        NumberFormat moneyFormat = NumberFormat.getCurrencyInstance( Locale.US );
```
Line 17 in method `main` declares three `double` variables and initializes two of them—`principal` to `1000.0` and `rate` to `0.05`. Java treats floating-point constants, like `1000.0` and `0.05` in Fig. 5.6, as type `double`. Similarly, Java treats whole number constants, like `7` and `-22`, as type `int`. Lines 21–22 declare `NumberFormat` reference `moneyFormat` and initialize it by calling `static` method `getCurrencyInstance` of class `NumberFormat`. This method returns a `NumberFormat` object that can format numeric values as currency (e.g., in the United States, currency values normally are preceded with a dollar sign, $). The argument to the method—`Locale.US`—indicates that
the currency values should be displayed starting with a dollar sign ($), use a decimal point to separate dollars and cents and use a comma to delineate thousands (e.g., $1,234.56). Class **Locale** provides constants that can be used to customize this program to represent currency values for other countries, so that currency formats are displayed properly for each locale (i.e., each country’s local-currency format). Class **NumberFormat** (imported at line 5) is located in package **java.text**, and class **Locale** (imported at line 6) is located in package **java.util**.

Line 25 declares **JTextArea** reference **outputTextArea** and initializes it with a new object of class **JTextArea** (from package **javax.swing**). A **JTextArea** is a GUI component that can display many lines of text. The message dialog that displays the **JTextArea** determines the width and height of the **JTextArea**, based on the **String** it contains. We introduce this GUI component now because we will see many examples throughout the text in which the program outputs contain too many lines to display on the screen. This GUI component allows us to scroll through the lines of text so we can see all the program output. The methods for placing text in a **JTextArea** include `setText` and `append`.

Line 28 uses **JTextArea** method `setText` to place a **String** in the **JTextArea** to which `outputTextArea` refers. Initially, a **JTextArea** contains an empty **String** (i.e., a **String** with no characters in it). The preceding statement replaces the empty **String** with one containing the column heads for our two columns of output—"Year" and "Amount on Deposit." The column heads are separated with a tab character (escape sequence `\t`). Also, the string contains the newline character (escape sequence `\n`), indicating that any additional text appended to the **JTextArea** should begin on the next line.

The `for` structure (lines 31–40) executes its body 10 times, varying control variable **year** from 1 to 10 in increments of 1. (Note that **year** represents **n** in the statement of the problem.) Java does not include an exponentiation operator. Instead, we use static method `pow` of class **Math** for this purpose. **Math.pow(x, y)** calculates the value of **x** raised to the **y**th power. Method `pow` takes two arguments of type **double** and returns a **double** value. Line 34 performs the calculation from the statement of the problem,

\[
a = p \times (1 + r)^n
\]

where **a** is **amount**, **p** is **principal**, **r** is **rate** and **n** is **year**.

Lines 37–38 `append` more text to the end of the **outputTextArea**. The text includes the current value of **year**, a tab character (to position to the second column), the result of the method call `moneyFormat.format(amount)`—which formats the **amount** as U. S. currency—and a newline character (to position the cursor in the **JTextArea** at the beginning of the next line).

Lines 43–44 display the results in a message dialog. Until now, the message displayed has always been a **String**. In this example, the second argument is **outputTextArea**—a GUI component. An interesting feature of class **JOptionPane** is that the message it displays with `showMessageDialog` can be a **String** or a GUI component, such as a **JTextArea**. In this example, the message dialog sizes itself to accommodate the **JTextArea**. We use this technique several times early in this chapter to display large text-based outputs. Later in this chapter, we demonstrate how to add a scrolling capability to the **JTextArea**, so the user can view a program’s output that is too large to display in full on the screen.
Notice that the variables *amount*, *principal* and *rate* are of type *double*. We did this for simplicity, because we are dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, this setting can cause trouble. Here is a simple explanation of what can go wrong when using *float* or *double* to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point): Two *double* dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your printout could appear as

```
14.23
+ 18.67
-------
32.91
```

but a person adding the individual numbers as printed would expect the sum to be 32.90. You have been warned!

**Good Programming Practice 5.10**

*Do not use variables of type *float* or *double* to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. In the exercises, we explore the use of integers to perform monetary calculations.* [Note: Some third-party vendors provide for-sale class libraries that perform precise monetary calculations.]

Note that the body of the *for* structure contains the calculation *1.0 + rate*, which appears as an argument to the *Math.pow* method. In fact, this calculation produces the same result each time through the loop, so repeating the calculation every iteration of the loop is wasteful.

**Performance Tip 5.1**

*Avoid placing expressions whose values do not change inside loops. But even if you do, many of today's sophisticated optimizing compilers will place such expressions outside loops in the generated compiled code.*

**Performance Tip 5.2**

*Many compilers contain optimization features that improve the code that you write, but it is still better to write good code from the start.*

### 5.5 The *switch* Multiple-Selection Structure

We have discussed the *if* single-selection structure and the *if/else* double-selection structure. Occasionally, an algorithm contains a series of decisions in which the algorithm tests a variable or expression separately for each of the constant integral values (i.e., values of types *byte*, *short*, *int* and *char*) the variable or expression may assume and takes different actions based on those values. Java provides the *switch* multiple-selection structure to handle such decision making. The applet of Fig. 5.7 demonstrates drawing lines, rectangles or ovals, based on an integer the user inputs via an input dialog.
// Fig. 5.7: SwitchTest.java
// Drawing lines, rectangles or ovals based on user input.

// Java core packages
import java.awt.Graphics;

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

public class SwitchTest extends JApplet {
    int choice;  // user's choice of which shape to draw

    // initialize applet by obtaining user's choice
    public void init() {
        String input;  // user's input

        // obtain user's choice
        input = JOptionPane.showInputDialog("Enter 1 to draw lines
" + "Enter 2 to draw rectangles\n" + "Enter 3 to draw ovals\n");

        // convert user's input to an int
        choice = Integer.parseInt( input );
    }

    // draw shapes on applet's background
    public void paint( Graphics g ) {
        // call inherited version of method paint
        super.paint( g );

        // loop 10 times, counting from 0 through 9
        for ( int i = 0; i < 10; i++ ) {
            // determine shape to draw based on user's choice
            switch ( choice ) {
                case 1:
                    g.drawLine( 10, 10, 250, 10 + i * 10 );
                    break;  // done processing case
                case 2:
                    g.drawRect( 10 + i * 10, 10 + i * 10,
                                50 + i * 10, 50 + i * 10 );
                    break;  // done processing case
                case 3:
                    g.drawOval( 10 + i * 10, 10 + i * 10,
                                 50 + i * 10, 50 + i * 10 );
                    break;  // done processing case
            }
        }
    }
}

Fig. 5.7  An example using switch (part 1 of 3).
default:
g.drawString( "Invalid value entered",
    10, 20 + i * 15 );
}
}  // end switch structure
}  // end for structure
}  // end paint method
}  // end class SwitchTest
Line 11 in applet SwitchTest defines instance variable choice of type int. This variable stores the user's input that determines which type of shape to draw in paint.

Method init (lines 14–26) declares local variable input of type String in line 16. This variable stores the String the user types in the input dialog. Lines 19–22 display the input dialog with static method JOptionPane.showMessageDialog and prompt the user to enter 1 to draw lines, 2 to draw rectangles or 3 to draw ovals. Line 25 converts input from a String to an int and assigns the result to choice.

Method paint (lines 29–62) contains a for structure (lines 35–60) that loops 10 times. In this example, the for structure’s header, in line 35, uses zero-based counting. The values of i for the 10 iterations of the loop are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, and the loop terminates when i’s value becomes 10. [Note: As you know, the applet container calls method paint after methods init and start. The applet container also calls method paint whenever the applet’s screen area must be refreshed—e.g., after another window that covered the applet’s area is moved to a different location on the screen.]

Nested in the for structure’s body is a switch structure (lines 38–58) that draws shapes based on the integer value input by the user in method init. The switch structure consists of a series of case labels and an optional default case.

When the flow of control reaches the switch structure, the program evaluates the controlling expression (choice) in the parentheses following keyword switch. The program compares the value of the controlling expression (which must evaluate to an integral value of type byte, char, short or int) with each case label. Assume that the user entered the integer 2 as his or her choice. The program compares 2 with each case in the switch. If a match occurs (case 2:), the program executes the statements for that case. For the integer 2, lines 44–47 draw a rectangle, using four arguments, representing the upper left x-coordinate, upper left y-coordinate, width and height of the rectangle, and the switch structure exits immediately with the break statement. Then, the program increments the counter variable in the for structure and reevaluates the loop-continuation condition to determine whether to perform another iteration of the loop.

The break statement causes program control to proceed with the first statement after the switch structure. (In this case, we reach the end of the for structure’s body, so con-
control flows to the control variable’s increment expression in the header of the `for` structure.) Without `break`, the `case`s in a `switch` statement would run together. Each time a match occurs in the structure, the statements for all the remaining `case`s will execute. (This feature is perfect for programming the iterative song “The Twelve Days of Christmas.”) If no match occurs between the controlling expression’s value and a `case` label, the `default` case executes, and the program draws an error message on the applet.

Each `case` can have multiple actions. The `switch` structure differs from other structures in that it does not require braces around multiple actions in each `case`. Figure 5.8 shows the general `switch` structure flowchart (using a `break` in each `case`). [Note: As an exercise, make a flowchart of the general `switch` structure without `breaks`.]

The flowchart makes it clear that each `break` statement at the end of a `case` causes control to exit the `switch` structure immediately. The `break` statement is not required for the last `case` in the `switch` structure (or the `default` case, when it appears last), because the program continues with the next statement after the `switch`.

Again, note that, besides small circles and arrows, the flowchart contains only rectangle and diamond symbols. It is the programmer’s responsibility to fill the rectangles and diamonds with actions and decisions appropriate to the algorithm. Although nested control structures are common, it is rare to find nested `switch` structures in a program.

**Fig. 5.8** The `switch` multiple-selection structure.
Common Programming Error 5.7

Forgetting a break statement when one is needed in a switch structure is a logic error.

Good Programming Practice 5.11

Provide a default case in switch statements. Cases not explicitly tested in a switch statement without a default case are ignored. Including a default case focuses the programmer on the need to process exceptional conditions. There are situations in which no default processing is needed.

Good Programming Practice 5.12

Although the cases and the default case in a switch structure can occur in any order, it is considered a good programming practice to place the default clause last.

Good Programming Practice 5.13

In a switch structure, when the default clause is listed last, the break for that case statement is not required. Some programmers include this break for clarity and symmetry with other cases.

Note that listing case labels together (such as case1: case2: with no statements between the cases) performs the same set of actions for each case.

When using the switch structure, remember that the expression after each case can be only a constant integral expression (i.e., any combination of character constants and integer constants that evaluates to a constant integer value). A character constant is represented as the specific character in single quotes, such as 'A'. An integer constant is simply an integer value. The expression after each case also can be a constant variable—i.e., a variable that contains a value which does not change for the entire program. Such a variable is declared with keyword final (discussed in Chapter 6). When we discuss object-oriented programming in Chapter 9, we present a more elegant way to implement switch logic. We use a technique called polymorphism to create programs that are often clearer, easier to maintain and easier to extend than programs using switch logic.

5.6 The do/while Repetition Structure

The do/while repetition structure is similar to the while structure. In the while structure, the program tests the loop-continuation condition at the beginning of the loop, before performing the body of the loop. The do/while structure tests the loop-continuation condition after performing the body of the loop; therefore, the loop body always executes at least once. When a do/while structure terminates, execution continues with the statement after the while clause. Note that it is not necessary to use braces in the do/while structure if there is only one statement in the body. However, most programmers include the braces, to avoid confusion between the while and do/while structures. For example,

```java
while ( condition )
```

normally is the first line of a while structure. A do/while structure with no braces around a single-statement body appears as

```java
do
  statement
while ( condition );
```
which can be confusing. Reader may misinterpret the last line—\texttt{while( condition );}—
as a \texttt{while} structure containing an empty statement (the semicolon by itself). Thus, to
avoid confusion, the \texttt{do/while} structure with one statement often is written as follows:

\begin{verbatim}
do {
  statement
} while ( condition );
\end{verbatim}

\textbf{Good Programming Practice 5.14}

Some programmers always include braces in a \texttt{do/while} structure, even if the braces are
not necessary. This helps eliminate ambiguity between the \texttt{while} structure and the \texttt{do/while} structure containing only one statement.

\textbf{Common Programming Error 5.8}

Infinite loops occur when the loop-continuation condition in a \texttt{while}, \texttt{for} or \texttt{do/while}
structure never becomes \texttt{false}. To prevent this situation, make sure that there is not a semi-
colon immediately after the header of a \texttt{while} or \texttt{for} structure. In a counter-controlled
loop, ensure that the control variable is incremented (or decremented) in the body of the
loop. In a sentinel-controlled loop, ensure that the sentinel value is eventually input.

The applet in Fig. 5.9 uses a \texttt{do/while} structure to draw 10 nested circles, using
\texttt{Graphics} method \texttt{drawOval}.

\begin{verbatim}
public class DoWhileTest extends JApplet {

  public void paint( Graphics g )
  {
    // call inherited version of method paint
    super.paint( g );

    int counter = 1;

    do {
      g.drawOval( 110 - counter * 10, 110 - counter * 10,
                  counter * 20, counter * 20 );
      ++counter;
    } while ( counter <= 10 );  // end do/while structure

  }  // end method paint

} // end class DoWhileTest
\end{verbatim}

\textbf{Fig. 5.9} Using the \texttt{do/while} repetition structure (part 1 of 2).
In method `paint` (lines 13–26), line 18 declares control variable `counter` and initializes it to 1. Upon entering the `do/while` structure, lines 21–22 send the `drawOval` message to the `Graphics` object to which `g` refers. The four arguments that represent the upper left x-coordinate, upper left y-coordinate, width and height of the oval’s bounding box (an imaginary rectangle in which the oval touches the center of all four sides of the rectangle) are calculated based on the value of `counter`. The program draws the innermost oval first. The bounding box’s upper left corner for each subsequent oval moves closer to the upper left corner of the applet. At the same time, the width and height of the bounding box are increased, to ensure that each new oval contains all the previous ovals. Line 23 increments `counter`. Then, the program evaluates the loop-continuation test at the bottom of the loop. The `do/while` flowchart in Fig. 5.10 makes it clear that the program does not evaluate the loop-continuation condition until after the action executes once.
5.7 Statements break and continue

The `break` and `continue` statements alter the flow of control. The `break` statement, when executed in a `while`, `for`, `do/while` or `switch` structure, causes immediate exit from that structure. Execution continues with the first statement after the structure. Common uses of the `break` statement are to escape early from a loop or skip the remainder of a `switch` structure (as in Fig. 5.7). Figure 5.11 demonstrates the `break` statement in a `for` repetition structure.

```java
// Fig. 5.11: BreakTest.java
// Using the break statement in a for structure
import javax.swing.JOptionPane;

public class BreakTest {
    public static void main( String args[] )
    {
        String output = "";
        int count;

        // loop 10 times
        for ( count = 1; count <= 10; count++ ) {
            // if count is 5, terminate loop
            if ( count == 5 )
                break; // break loop only if count == 5

            output += count + " ";
        }  // end for structure

        output += "\nBroke out of loop at count = " + count;
        JOptionPane.showMessageDialog( null, output );

        System.exit( 0 ); // terminate application
    }  // end method main
} // end class BreakTest
```

Fig. 5.11  Using the break statement in a for structure.
When the if structure at line 19 in the for structure detects that count is 5, the break statement at line 20 executes. This statement terminates the for structure, and the program proceeds to line 26 (immediately after the for). Line 26 completes the string to display in a message dialog at line 27. The loop fully executes its body only four times.

The continue statement, when executed in a while, for or do/while structure, skips the remaining statements in the loop body and proceeds with the next iteration of the loop. In while and do/while structures, the program evaluates the loop-continuation test immediately after the continue statement executes. In for structures, the increment expression executes, then the program evaluates the loop-continuation test. Earlier, we stated that the while structure could be used in most cases to represent the for structure. The one exception occurs when the increment expression in the while structure follows the continue statement. In this case, the increment does not execute before the program evaluates the repetition-continuation condition, so the while structure does not execute in the same manner as does the for structure. Figure 5.12 uses the continue statement in a for structure to skip the string concatenation statement (line 22) when the if structure (line 18) determines that the value of count is 5. When the continue statement executes, program control continues with the increment of the control variable in the for structure.

Good Programming Practice 5.15

Some programmers feel that break and continue violate structured programming. Because the effects of these statements are achievable with structured programming techniques, these programmers do not use break and continue.

```java
// Fig. 5.12: ContinueTest.java
// Using the continue statement in a for structure

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

public class ContinueTest {

    public static void main( String args[] ) {

        String output = "";
        // loop 10 times
        for ( int count = 1; count <= 10; count++ ) {

            // if count is 5, continue with next iteration of loop
            if ( count == 5 )
                continue; // skip remaining code in loop

            // only if count == 5
            output += count + " ";

        } // end for structure

    }
}
```

Fig. 5.12 Using the continue statement in a for structure (part 1 of 2).
Performance Tip 5.3

The `break` and `continue` statements, when used properly, perform faster than the corresponding structured techniques.

Software Engineering Observation 5.1

There is a tension between achieving quality software engineering and achieving the best performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

5.8 Labeled `break` and `continue` Statements

The `break` statement can break out of only an immediately enclosing `while`, `for`, `do/while` or `switch` structure. To break out of a nested set of structures, you can use the labeled `break` statement. This statement, when executed in a `while`, `for`, `do/while` or `switch` structure, causes immediate exit from that structure and any number of enclosing repetition structures; program execution resumes with the first statement after the enclosing labeled block (i.e., a set of statements enclosed in curly braces and preceded by a label). The block can be either a repetition structure (the body would be the block) or a block in which the repetition structure is the first executable code. Labeled `break` statements are commonly used to terminate nested looping structures containing `while`, `for`, `do/while` or `switch` structures. Figure 5.13 demonstrates the labeled `break` statement in a nested `for` structure.

The block (lines 14–37) begins with a `label` (an identifier followed by a colon) at line 14; here, we use the label “stop:.” The block is enclosed in braces at the end of line 14 and line 37 and includes the nested `for` structure (lines 17–32) and the string-concatenation statement at line 35. When the `if` structure at line 23 detects that `row` is equal to 5, the `break` statement at line 24 executes. This statement terminates both the `for` structure at line 20 and its enclosing `for` structure at line 17. The program proceeds immediately to...
line 39—the first statement after the labeled block. The outer for structure fully executes its body only four times. Notice that the string-concatenation statement at line 35 never executes, because it is in the labeled block’s body, and the outer for structure never completes.

```
1 // Fig. 5.13: BreakLabelTest.java
2 // Using the break statement with a label
3
4 // Java extension packages
5 import javax.swing.JOptionPane;
6
7 public class BreakLabelTest {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         String output = ""
13
14         stop: {   // labeled block
15
16             // count 10 rows
17             for ( int row = 1; row <= 10; row++ ) {
18
19                 // count 5 columns
20                 for ( int column = 1; column <= 5 ; column++ ) {
21
22                     // if row is 5, jump to end of "stop" block
23                     if ( row == 5 )
24                         break stop; // jump to end of stop block
25
26                     output += "*  ";
27
28                 } // end inner for structure
29
30                 output += "\n";
31
32             } // end outer for structure
33
34             // the following line is skipped
35             output += "\nLoops terminated normally";
36
37         } // end labeled block
38
39         JOptionPane.showMessageDialog(
40             null, output,"Testing break with a label",
41             JOptionPane.INFORMATION_MESSAGE );
42
43         System.exit( 0 ); // terminate application
44
45     } // end method main
46
47 } // end class BreakLabelTest
```

Fig. 5.13 Using a labeled break statement in a nested for structure (part 1 of 2).
The `continue` statement proceeds with the next iteration (repetition) of the immediately enclosing `while`, `for` or `do/while` structure. The labeled `continue` statement, when executed in a repetition structure (`while`, `for` or `do/while`), skips the remaining statements in that structure’s body and any number of enclosing repetition structures and proceeds with the next iteration of the enclosing labeled repetition structure (i.e., a repetition structure preceded by a label). In labeled `while` and `do/while` structures, the program evaluates the loop-continuation test immediately after the `continue` statement executes. In a labeled `for` structure, the increment expression is executed, and then the loop-continuation test is evaluated. Figure 5.14 uses the labeled `continue` statement in a nested `for` structure to enable execution to continue with the next iteration of the outer `for` structure.

The labeled `for` structure (lines 14–32) starts at the `nextRow` label. When the `if` structure at line 24 in the inner `for` structure detects that `column` is greater than `row`, the `continue` statement at line 25 executes, and program control continues with the increment of the control variable of the outer `for` loop. Even though the inner `for` structure counts from 1 to 10, the number of * characters output on a row never exceeds the value of `row`.

**Performance Tip 5.4**
The program in Fig. 5.14 can be made simpler and more efficient by replacing the condition in the `for` structure at line 21 with `column <= row` and removing the `if` structure at lines 24–25 from the program.

### 5.9 Logical Operators

So far, we have studied only simple conditions, such as `count <= 10`, `total > 1000` and `number != sentinelValue`. These conditions were expressed in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`. Each decision tested one condition. To test multiple conditions in the process of making a decision, we performed these tests in separate statements or in nested `if` or `if/else` structures.

Java provides logical operators to enable programmers to form more complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `&` (boolean logical AND), `||` (logical OR), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR) and `!` (logical NOT, also called logical negation).
// Fig. 5.14: ContinueLabelTest.java
// Using the continue statement with a label

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

public class ContinueLabelTest {

    // main method begins execution of Java application
    public static void main( String args[] )
    {
        String output = "";

        nextRow:   // target label of continue statement

            // count 5 rows
            for ( int row = 1; row <= 5; row++ ) {
                output += "\n";

                // count 10 columns per row
                for ( int column = 1; column <= 10; column++ ) {
                    // if column greater than row, start next row
                    if ( column > row )
                        continue nextRow; // next iteration of
                                           // labeled loop

                    output += "*  ";
                }  // end inner for structure

            }  // end outer for structure

            JOptionPane.showMessageDialog(
               null, output,"Testing continue with a label",
               JOptionPane.INFORMATION_MESSAGE );

            System.exit( 0 ); // terminate application
    }  // end method main

} // end class ContinueLabelTest

Fig. 5.14 Using a labeled continue statement in a nested for structure.
Suppose we wish to ensure at some point in a program that two conditions are both true before we choose a certain path of execution. In this case, we can use the logical && operator, as follows:

```java
if ( gender == 1 && age >= 65 )
++seniorFemales;
```

This if statement contains two simple conditions. The condition `gender == 1` might be evaluated, for example, to determine if a person is a female. The condition `age >= 65` is evaluated to determine if a person is a senior citizen. The two simple conditions are evaluated first, because the precedences of == and >= are both higher than the precedence of &&. The if statement then considers the combined condition

```java
gender == 1 && age >= 65
```

This condition is true if and only if both of the simple conditions are true. If this combined condition is indeed true, the if structure’s body statement increments variable `seniorFemales` by 1. If either or both of the simple conditions are false, the program skips the increment and proceeds to the statement following the if structure. The preceding combined condition can be made more readable by adding redundant parentheses:

```java
( gender == 1 ) && ( age >= 65 )
```

The table in Fig. 5.15 summarizes the && operator. The table shows all four possible combinations of false and true values for expression1 and expression2. Such tables are often called truth tables. Java evaluates to false or true all expressions that include relational operators, equality operators and/or logical operators.

Now let us consider the || (logical OR) operator. Suppose we wish to ensure that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the || operator, as in the following program segment:

```java
if ( semesterAverage >= 90 || finalExam >= 90 )
System.out.println ( "Student grade is A" );
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` evaluates to determine if the student deserves an “A” in the course because of a solid performance throughout the semester. The condition `finalExam >= 90` evaluates to determine if the student deserves an “A” in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 &amp;&amp; expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Fig. 5.15 Truth table for the && (logical AND) operator.
and awards the student an “A” if either or both of the simple conditions are true. Note that
the only time the message “Student grade is A” is not printed is when both of the simple
conditions are false. Figure 5.16 is a truth table for the logical OR operator (||).

The && operator has a higher precedence than the || operator. Both operators associate from left to right. An expression containing && or || operators is evaluated only until truth or falsity is known. Thus, evaluation of the expression

gender == 1 && age >= 65

stops immediately if gender is not equal to 1 (i.e., the entire expression is false) and
continues if gender is equal to 1 (i.e., the entire expression could still be true if the condition
age >= 65 is true). This performance feature for evaluation of logical AND and
logical OR expressions is called short-circuit evaluation.

**Common Programming Error 5.9**

In expressions using operator &&, it is possible that a condition—we will call this the depen-
dent condition—may require another condition to be true for it to be meaningful to evalu-
ate the dependent condition. In this case, the dependent condition should be placed after the
other condition, or an error might occur.

**Performance Tip 5.5**

In expressions using operator &&, if the separate conditions are independent of one another,
make the condition that is most likely to be false the leftmost condition. In expressions us-
ing operator ||, make the condition that is most likely to be true the leftmost condition.
This can reduce a program’s execution time.

The boolean logical AND (&) and boolean logical inclusive OR (||) operators work
identically to the regular logical AND and logical OR operators, with one exception: The
boolean logical operators always evaluate both of their operands (i.e., there is no short-cir-
cuit evaluation). Therefore, the expression

gender == 1 & age >= 65

evaluates age >= 65 regardless of whether gender is equal to 1. This method is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator
has a required side effect—a modification of a variable’s value. For example, the expres-

| expression1 | expression2 | expression1 || expression2 |
|-------------|-------------|-------------|
| false       | false       | false       |
| false       | true        | true        |
| true        | false       | true        |
| true        | true        | true        |

**Fig. 5.16** Truth table for the || (logical OR) operator.
guarantees that the condition \texttt{++age \geq 65} will be evaluated. Thus, the variable \texttt{age} is incremented in the preceding expression, regardless of whether the overall expression is \texttt{true} or \texttt{false}.

\textbf{Good Programming Practice 5.16}

For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they are often more trouble than they are worth.

A condition containing the boolean logical exclusive OR (~\texttt{\^}~) operator is \texttt{true} if and only if one of its operands results in a \texttt{true} value and one results in a \texttt{false} value. If both operands are \texttt{true} or both are \texttt{false}, the result of the entire condition is \texttt{false}. Figure 5.17 is a truth table for the boolean logical exclusive OR operator (~\texttt{\^}~). This operator is also guaranteed to evaluate both of its operands (i.e., there is no short-circuit evaluation).

Java provides the \texttt{!} (logical negation) operator to enable a programmer to “reverse” the meaning of a condition. Unlike the logical operators \texttt{\&\&}, \texttt{\&}, \texttt{||}, \texttt{|}, \texttt{\^}, which combine two conditions (i.e., they are binary operators), the logical negation operator has only a single condition as an operand (i.e., they are unary operator). The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is \texttt{false}, such as in the following program segment:

\begin{verbatim}
if ( ! ( grade == sentinelValue ) )
    System.out.println( "The next grade is " + grade );
\end{verbatim}

The parentheses around the condition \texttt{grade == sentinelValue} are needed, because the logical negation operator has a higher precedence than the equality operator. Figure 5.18 is a truth table for the logical negation operator.

In most cases, the programmer can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written as follows:

\begin{verbatim}
if ( grade != sentinelValue )
    System.out.println( "The next grade is " + grade );
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\texttt{expression1} & \texttt{expression2} & \texttt{expression1 \^ expression2} \\
\hline
\texttt{false} & \texttt{false} & \texttt{false} \\
\texttt{false} & \texttt{true} & \texttt{true} \\
\texttt{true} & \texttt{false} & \texttt{true} \\
\texttt{true} & \texttt{true} & \texttt{false} \\
\hline
\end{tabular}
\caption{Truth table for the boolean logical exclusive OR (~\texttt{\^}~) operator.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\texttt{expression} & \texttt{!expression} \\
\hline
\texttt{false} & \texttt{true} \\
\texttt{true} & \texttt{false} \\
\hline
\end{tabular}
\caption{Truth table for operator \texttt{!} (logical negation, or logical NOT).}
\end{table}
if ( grade != sentinelValue )
    System.out.println( "The next grade is " + grade );

This flexibility can help a programmer express a condition in a more convenient manner.

The application in Fig. 5.19 demonstrates all of the logical operators and boolean logical operators by producing their truth tables. The program uses string concatenation to create the string that is displayed in a JTextArea.

```java
// Fig. 5.19: LogicalOperators.java
// Demonstrating the logical operators
// Java extension packages
import javax.swing.*;

public class LogicalOperators {

    // main method begins execution of Java application
    public static void main( String args[] )
    {
        // create JTextArea to display results
        JTextArea outputArea = new JTextArea( 17, 20 );

        // attach JTextArea to a JScrollPane so user can
        // scroll through results
        JScrollPane scroller = new JScrollPane( outputArea );

        String output;

        // create truth table for && operator
        output = "Logical AND (&&)" + 
                "false && false: " + ( false && false ) + 
                "false && true: " + ( false && true ) + 
                "true && false: " + ( true && false ) + 
                "true && true: " + ( true && true );

        // create truth table for || operator
        output += 
                "Logical OR (||)
                "false || false: " + ( false || false ) + 
                "false || true: " + ( false || true ) + 
                "true || false: " + ( true || false ) + 
                "true || true: " + ( true || true );

        // create truth table for & operator
        output += 
                "Boolean logical AND (&)
                "false & false: " + ( false & false ) + 
                "false & true: " + ( false & true ) + 
                "true & false: " + ( true & false ) + 
                "true & true: " + ( true & true );

        // create truth table for | operator
        output += 
                "Boolean logical inclusive OR (|)
                "false | false: " + ( false | false ) + 
                "false | true: " + ( false | true ) + 
                "true | false: " + ( true | false ) + 
                "true | true: " + ( true | true );

        // create truth table for & operator
        output += 
                "Boolean logical AND (&)
                "false & false: " + ( false & false ) + 
                "false & true: " + ( false & true ) + 
                "true & false: " + ( true & false ) + 
                "true & true: " + ( true & true );

        // create truth table for | operator
        output += 
                "Boolean logical inclusive OR (|)
                "false | false: " + ( false | false ) + 
                "false | true: " + ( false | true ) + 
                "true | false: " + ( true | false ) + 
                "true | true: " + ( true | true );
    }
}

Fig. 5.19 Demonstrating the logical operators (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/2/01
In the output of Fig. 5.19, the strings “true” and “false” indicate true and false for the operands in each condition. The result of the condition is shown as true or false. Note that when you concatenate a boolean value with a String, Java automatically adds the string “false” or “true,” based on the boolean value.

Line 13 in method main creates a JTextArea. The numbers in the parentheses indicate that the JTextArea has 17 rows and 20 columns. Line 17 declares JScrollPane reference scroller and initializes it with a new JScrollPane object. Class

```
// create truth table for ^ operator
output += "\n\nBoolean logical exclusive OR (^)"
"\ntrue ^ false: " + (true | false) +
"\ntrue ^ true: " + (true | true);

// create truth table for ! operator
output += "\n\nLogical NOT (!)"
"!false: " + (!false) +
"!true: " + (!true);

outputArea.setText(output);  // place results in JTextArea
JOptionPane.showMessageDialog(null, scroller,  "Truth Tables", JOptionPane.INFORMATION_MESSAGE);
System.exit(0);  // terminate application
```
**JScrollPane** (from package `javax.swing`) provides a GUI component with scrolling functionality. A **JScrollPane** object is initialized with the GUI component for which it will provide scrolling functionality (**outputArea** in this example). This initialization attaches the GUI component to the **JScrollPane**. When you execute this application, notice the **scrollbar** on the right side of the **JTextArea**. You can click the arrows at the top or bottom of the scrollbar to scroll up or down, respectively, through the text in the **JTextArea** one line at a time. You can also drag the **scroll box** (also called the **thumb**) up or down to scroll through the text rapidly.

Lines 22–59 build the **output** string that is displayed in the **outputArea**. Line 61 uses method **setText** to replace the text in **outputArea** with the **output** string. Lines 63–64 display a message dialog. The second argument, **scroller**, indicates that the **scroller** (and the **outputArea** attached to it) should be displayed as the message in the message dialog.

The chart in Fig. 5.20 shows the precedence and associativity of the Java operators introduced up to this point. The operators are shown from top to bottom in decreasing order of precedence.

### 5.10 Structured Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture is, and our collective wisdom is considerably sparser. We have learned that structured programming produces programs that are easier than unstructured programs to understand and hence are easier to test, debug, modify and even prove correct in a mathematical sense.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
<td>parentheses</td>
</tr>
<tr>
<td>++ --</td>
<td>right to left</td>
<td>unary postfix</td>
</tr>
<tr>
<td>++ -- + - ! (type)</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
<td>boolean logical AND</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
<td>boolean logical exclusive OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

**Fig. 5.20** Precedence and associativity of the operators discussed so far.
Figure 5.21 summarizes Java’s control structures. Small circles are used in the figure to indicate the single entry and single exit points of each structure. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control structures, and to build structured programs by properly combining control structures in two simple ways.
For simplicity, only single-entry/single-exit control structures are used; there is only one way to enter and only one way to exit each control structure. Connecting control structures in sequence to form structured programs is simple: The exit point of one control structure is connected to the entry point of the next control structure (i.e., the control structures are simply placed one after another in a program); we have called this method “control-structure stacking.” The rules for forming structured programs also allow for control-structure nesting.

Figure 5.22 shows the rules for forming properly structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action, including input/output.

Applying the rules of Fig. 5.22 always results in a structured flowchart with a neat, building-block-like appearance (Fig. 5.23). For example, repeatedly applying Rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence (Fig. 5.24). Rule 2 generates a stack of control structures; so let us call Rule 2 the stacking rule. [Note: The symbols at the top and bottom of Fig. 5.23 represent the beginning and end of a program, respectively.]

Rule 3 is called the nesting rule. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control structures. For example, in Fig. 5.25, the rectangle in the simplest flowchart is first replaced with a double-selection (if/else) structure. Then, Rule 3 is applied again to both of the rectangles in the double-selection structure, replacing each of the rectangles with double-selection structures. The dashed boxes around each of the double-selection structures represent the rectangle that was replaced with a double-selection structure.

### Rules for Forming Structured Programs

1. Begin with the “simplest flowchart” (Fig. 5.23).
2. Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3. Any rectangle (action) can be replaced by any control structure (sequence, if, if/else, switch, while, do/while or for).
4. Rules 2 and 3 may be applied as often as you like and in any order.

---

**Fig. 5.22** Rules for forming structured programs.

**Fig. 5.23** The simplest flowchart.
Fig. 5.24  Repeatedly applying Rule 2 of Fig. 5.22 to the simplest flowchart.

Fig. 5.25  Applying Rule 3 of Fig. 5.22 to the simplest flowchart.
Rule 4 generates larger, more involved and more deeply nested structures. The flowcharts that emerge from applying the rules in Fig. 5.22 constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

The beauty of the structured approach is that we use only seven simple single-entry/single-exit pieces and we assemble them in only two simple ways. Figure 5.26 shows the kinds of stacked building blocks that emerge from applying Rule 2 and the kinds of nested building blocks that emerge from applying Rule 3. The figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the \textbf{goto} statement).

If the rules in Fig. 5.22 are followed, you cannot create an unstructured flowchart (such as that in Fig. 5.27). If you are uncertain if a particular flowchart is structured, apply the rules of Fig. 5.22 in reverse to try to reduce the flowchart to the simplest flowchart. If the flowchart is reducible to the simplest flowchart, the original flowchart is structured; otherwise, it is not.

Structured programming promotes simplicity. Bohm and Jacopini have shown that only three forms of control are needed—sequence, selection and repetition.
Sequence is trivial. Selection is implemented in one of three ways:

- an `if` structure (single selection),
- an `if/else` structure (double selection), or
- a `switch` structure (multiple selection).

In fact, it is straightforward to prove that the `if` structure is sufficient for any form of selection; everything that can be done with the `if/else` structure and the `switch` structure can be implemented by combining `if` structures (although perhaps not as elegantly).

Repetition is implemented in one of three ways:

- a `while` structure,
- a `do/while` structure, or
- a `for` structure.

It is straightforward to prove that the `while` structure is sufficient to provide any form of repetition. Everything that can be done with the `do/while` structure and the `for` structure can be done with the `while` structure (although perhaps not as elegantly).

Combining these results illustrates that any form of control ever needed in a Java program can be expressed in terms of

- a sequence,
- an `if` structure (selection), or
- a `while` structure (repetition).

And these control structures can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

In this chapter, we have discussed how to compose programs from control structures containing actions and decisions. In Chapter 6, we introduce another program structuring unit, called the `method`. We will learn to compose large programs by combining methods that, in turn, are composed of control structures. We will also discuss how methods promote software reusability. In Chapter 8, we discuss in more detail Java’s other program-structuring unit, called the `class`. We will then create objects from classes and proceed with our treatment of object-oriented programming.

### 5.11 (Optional Case Study) Thinking About Objects: Identifying Objects’ States and Activities

In “Thinking About Objects,” Section 4.14, we determined many of the class attributes needed to implement the elevator simulator and added them to the class diagram of Fig. 4.18. In this section, we show how these attributes represent an object’s state, or condition. We identify the set of possible states that our objects may occupy and discuss how these objects change state in response to messages. We also discuss the workflow, or the activities, that an object performs in our elevator simulation.

**Statechart Diagrams**

Objects in a system have states. A state describes the condition of an object at a given time. Statechart diagrams (also called state diagrams) give us a way to express how, and under what conditions, the objects in a system change state. Unlike the class and object
diagrams presented in earlier case-study sections, statechart diagrams model the behavior of the system.

Figure 5.28 is a simple statechart diagram that models the states of an object of either class **FloorButton** or class **ElevatorButton**. The UML represents each state in a statechart diagram as a **rounded rectangle** with the name of the state placed inside the rectangle. A **solid circle** with an attached arrowhead designates the initial state (in this case, the “Not pressed” state). Notice that this statechart diagram models the **boolean** attribute **pressed** in the class diagram of Fig. 4.18. The attribute is initialized to **false**, or the “Not pressed” state according to the statechart diagram.

The arrows indicate **transitions** between states. An object can transition from one state to another in response to a **message**. For example, the **FloorButton** and **ElevatorButton** objects change from the “Not pressed” state to the “Pressed” state in response to a **buttonPressed** message, and the **pressed** attribute changes to a value of **true**. The name of the message that causes a transition is written near the line that corresponds to that transition. (We explain messages in Section 7.10 and Section 10.22.)

Objects from other classes, such as **Light**, **Elevator** and **Person**, have similar statechart diagrams. Class **Light** has an “on” state or an “off” state—transitions between these states occur as a result of “turn on” and “turn off” events, respectively. Class **Elevator** and class **Person** each have a “moving” state and a “waiting” state—transitions between these states occur as a result of “start moving” and “stop moving” events, respectively.

**Activity Diagrams**

The **activity diagram** is similar to the statechart diagram in that they both model aspects of system behavior. Unlike a statechart diagram, an activity diagram models an object’s workflow during program execution. An activity diagram is a flowchart that models the **actions** the object will perform and in what order. The activity diagram in Fig. 5.29 models the activities of a person. The diagram begins with the person moving toward the floor button. If the door is open, the person waits for the current elevator passenger (if one exists) to exit then enters the elevator. If the door is closed, the person presses the floor button and waits for the elevator to open the door. When the door opens, the person waits for the elevator passenger to exit (if one exists) then enters the elevator. The person presses the elevator button, which causes the elevator to move to the other floor, unless the elevator already services that floor; the person then waits for the doors to re-open and exits the elevator after the doors open.

---

![Statechart diagram for FloorButton and ElevatorButton objects.](image)

1. We use **multithreading** and **synchronized methods** in Section 15.12 to guarantee that the passenger riding the elevator will exit before the person waiting for the elevator will enter.
The UML represents activities as ovals in activity diagrams. The name of the activity is placed inside the oval. An arrow connects two activities, indicating the order in which the activities are performed. As with statechart diagrams, the solid circle indicates the starting activity. In this case, the person moving toward the floor button is the starting activity. The activity flow arrives at a branch (indicated by the small diamond symbol) after the person moves to the floor button. This point determines the next activity based on the associated guard condition (in square brackets above the transition), which states that the transition occurs if this condition is met. For example, in Fig. 5.29, if the floor door is closed, the person presses the floor button, waits for the door to open, waits for the passenger (if there is one) to exit the elevator, then enters the elevator. However, if the floor door is open, the person waits for the passenger (if there is one) to exit the elevator, then enters the elevator. Regardless of whether the floor door was open or closed at the last diamond symbol, the person now presses the elevator button (which causes the doors to close and the elevator to move to the other floor), the person waits for the elevator door to open—when this door opens, the person exits the elevator. Activity diagrams are similar to the flowcharts for control structures presented in Chapters 4 and 5—both diagram types employ diamond symbols to alter the flow of control between activities.

**Fig. 5.29** Activity diagram for a Person object.
Figure 5.30 shows an activity diagram for the elevator. The diagram begins when a button is pressed. If the button is an elevator button, the elevator sets `summoned` to false (we explain this `boolean` variable in a moment), closes the elevator door, moves to the other floor, resets the elevator button, rings the bell and opens the elevator door. If the button is a floor button, the next branch determines the next transition, based on whether the elevator is moving. If the elevator is idle, the next branch determines which floor button generated the request. If the request originated from the current floor on which the elevator is located, the elevator resets its button, rings its bell and opens its door. If the request originated from the opposite floor, the elevator closes the door and moves to the opposite (destination) floor, where the elevator resets its button, rings its bell and opens its door. Now consider the activity if the elevator is moving. A separate branch determines which floor button generated the request. If the request originated from the destination floor, the elevator continues traveling to that floor. If the request originated from the floor from which the elevator departed, the elevator continues traveling to the destination floor, but must remember to return to the requesting floor. The `summoned` attribute, originally displayed in Fig. 4.18, is set to `true` so that the elevator knows to return to the other floor after the elevator opens its door.
We have taken the first steps to modeling the behavior of the system and have shown how the attributes of an object determine that object’s activity. In “Thinking About Objects,” Section 6.17, we investigate the behaviors for all classes to give a more accurate interpretation of the system behavior by “filling in” the final compartment for the classes in our class diagram.

**SUMMARY**

- The **for** repetition structure handles all of the details of counter-controlled repetition. The general format of the **for** structure is

  ```java
  for (expression1; expression2; expression3) 
  statement
  ```

  where `expression1` initializes the loop’s control variable, `expression2` is the loop-continuation condition and `expression3` modifies the control variable, so that the loop-continuation condition eventually becomes `false`.

- A **JTextArea** is a GUI component that is capable of displaying many lines of text.

- Method `setText` replaces the text in a **JTextArea**. Method `append` adds text to the end of the text in a **JTextArea**.

- **NumberFormat** static method `getCurrencyInstance` returns a **NumberFormat** object that can format numeric values as currency. The argument **Locale.US** indicates that the currency values should be displayed starting with a dollar sign ($), use a decimal point to separate dollars and cents and use a comma to delineate thousands.

- Class **Locale** provides constants that can be used to customize programs to represent currency values for other countries, so that currency formats are displayed properly for each locale.

- Class **NumberFormat** is located in package **java.text**.

- Class **Locale** is located in package **java.util**.

- An interesting feature of class **JOptionPane** is that the message it displays with `showMessageDialog` can be a **String** or a GUI component, such as a **JTextArea**.

- The **switch** structure handles a series of decisions in which a particular variable or expression is tested for values it may assume, and different actions are taken. In most programs, it is necessary to include a **break** statement after the statements for each **case**. Several **cases** can execute the same statements by listing the **case** labels together before the statements. The **switch** structure can only test for constant integral expressions.

- The **do/while** repetition structure tests the loop-continuation condition at the end of the loop, so the body of the loop will be executed at least once. The format for the **do/while** structure is

  ```java
  do {
  statement
  } while (condition);
  ```

- The **break** statement, when executed in one of the repetition structures (**for**, **while** and **do/while**), causes immediate exit from the structure.

- The **continue** statement, when executed in one of the repetition structures (**for**, **while** and **do/while**), skips any remaining statements in the body of the structure and proceeds with the test for the next iteration of the loop.

- To break out of a nested set of structures, use the labeled **break** statement. This statement, when executed in a **while**, **for**, **do/while** or **switch** structure, causes immediate exit from that
structure and any number of enclosing repetition structures; program execution resumes with the first statement after the enclosing labeled block.

- The labeled `continue` statement, when executed in a repetition structure (while, for or do/while), skips the remaining statements in that structure’s body and any number of enclosing repetition structures and proceeds with the next iteration of the enclosing labeled repetition structure.

- Logical operators may be used to form complex conditions by combining conditions. The logical operators are `&&`, `&`, `||`, `|`, `^` and `!`, meaning logical AND, boolean logical AND, logical OR, boolean logical inclusive OR, boolean logical exclusive OR and logical NOT (negation), respectively.

- Class `JScrollPane` provides a GUI component with scrolling functionality.

**TERMINOLOGY**

| ! operator | labeled repetition structure |
| && operator | `Locale` class |
| || operator | `Locale.US` |
| `append` method of class `JTextArea` | logical AND (`&&`) |
| boolean logical AND (`&`) | logical negation (`!`) |
| boolean logical exclusive OR (`^`) | logical operators |
| boolean logical inclusive OR (`|`) | logical OR (`||`) |
| `break` | long |
| `case` label | loop-continuation condition |
| `continue` | multiple selection |
| counter-controlled repetition | nested control structures |
| `default` case in `switch` | `NumberFormat` class |
| definite repetition | off-by-one error |
| `do/while` repetition structure | repetition structures |
| `for` repetition structure | scroll box |
| infinite loop | scrollbar |
| `java.text` package | short-circuit evaluation |
| `java.util` package | single-entry/single-exit control structures |
| `JScrollPane` class | stacked control structures |
| `JTextArea` class | `switch` selection structure |
| labeled `break` statement | thumb of a scrollbar |
| labeled block | `while` repetition structure |

**SELF-REVIEW EXERCISES**

5.1 State whether each of the following is true or false. If false, explain why.

a) The `default` case is required in the `switch` selection structure.

b) The `break` statement is required in the default case of a `switch` selection structure.

c) The expression `(x > y && a < b)` is true if either `x > y` is true or `a < b` is true.

d) An expression containing the `||` operator is true if either or both of its operands is true.

5.2 Write a Java statement or a set of Java statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99, using a `for` structure. Assume that the integer variables `sum` and `count` have been declared.

b) Calculate the value of `2.5` raised to the power of `3`, using the `pow` method.

c) Print the integers from 1 to 20, using a `while` loop and the counter variable `x`. Assume that the variable `x` has been declared, but not initialized. Print only five integers per line.  
   [Hint: Use the calculation `x % 5`. When the value of this expression is 0, print a newline character; otherwise, print a tab character. Assume that this code is an application; use
the `System.out.println()` method to output the newline character, and use the `System.out.print( '\t' )` method to output the tab character.]

d) Repeat Exercise 5.2 (c), using a `for` structure.

5.3 Find the error in each of the following code segments, and explain how to correct it:

a) `x = 1;

    while ( x <= 10 );
    x++;
}

b) `for ( y = .1; y != 1.0; y += .1 )`

    `System.out.println( y );`

c) `switch ( n ) {
    case 1:
        System.out.println( "The number is 1" );
    case 2:
        System.out.println( "The number is 2" );
        break;
    default:
        System.out.println( "The number is not 1 or 2" );
        break;
}

d) The following code should print the values 1 to 10.

    n = 1;

    while ( n < 10 )
        System.out.println( n++ );

`ANSWERS TO SELF-REVIEW EXERCISES`

5.1 a) False. The `default` case is optional. If no default action is needed, then there is no need for a `default` case. b) False. The `break` statement is used to exit the `switch` structure. The `break` statement is not required for the last case in a `switch` structure. c) False. Both of the relational expressions must be true for the entire expression to be true when using the `&&` operator. d) True.

5.2 The answers to Exercise 5.2 are as follows:

a) `sum = 0;

    for ( count = 1; count <= 99; count += 2 )
        sum += count;

b) `Math.pow( 2.5, 3 )`

c) `x = 1;

    while ( x <= 20 ) {
        System.out.print( x );

        if ( x % 5 == 0 )
            System.out.println();
        else
            System.out.print( '\t' );

        ++x;
    }`
d) \[ \text{for ( } x = 1; x <= 20; x++ \text{ ) \{ } \]
\[ \text{System.out.print( } x \text{ ); } \]
\[ \text{if ( } x \% 5 == 0 \text{ ) } \]
\[ \text{System.out.println(); } \]
\[ \text{else } \]
\[ \text{System.out.print( '\t' ); } \]
\[ \text{}} \]

\[ \text{or} \]

\[ \text{for ( } x = 1; x <= 20; x++ \text{ ) } \]
\[ \text{if ( } x \% 5 == 0 \text{ ) } \]
\[ \text{System.out.println( } x \text{ ); } \]
\[ \text{else } \]
\[ \text{System.out.print( } x + "\t" \text{ ); } \]

5.3 The answers to Exercise 5.3 are as follows:

a) Error: The semicolon after the while header causes an infinite loop, and there is a missing left brace.
Correction: Replace the semicolon by a {, or remove both the ; and the }.

b) Error: Using a floating-point number to control a for repetition structure may not work, because floating-point numbers are represented only approximately by most computers.
Correction: Use an integer, and perform the proper calculation in order to get the values you desire:
\[ \text{for ( } y = 1; y != 10; y++ \text{ ) } \]
\[ \text{System.out.println( ( float ) y / 10 ); } \]

c) Error: Missing break statement in the statements for the first case.
Correction: Add a break statement at the end of the statements for the first case. Note that this omission is not necessarily an error if the programmer wants the statement of case2: to execute every time the case1: statement executes.

d) Error: Improper relational operator used in the while repetition-continuation condition.
Correction: Use <= rather than <, or change 10 to 11.

EXERCISES

5.4 Find the error(s) in each of the following segments of code:

a) \[ \text{for ( } x = 100; x >= 1; x++ \text{ ) } \]
\[ \text{System.out.println( } x \text{ ); } \]

b) The following code should print whether integer value is odd or even:
\[ \text{switch ( value \% 2 ) \{ } \]
\[ \text{case 0: } \]
\[ \text{System.out.println( "Even integer" ); } \]
\[ \text{case 1: } \]
\[ \text{System.out.println( "Odd integer" ); } \]
\[ \text{}} \]

c) The following code should output the odd integers from 19 to 1:
\[ \text{for ( } x = 19; x >= 1; x += 2 \text{ ) } \]
\[ \text{System.out.println( } x \text{ ); } \]
d) The following code should output the even integers from 2 to 100:

```java
counter = 2;

do {
    System.out.println( counter );
    counter += 2;
} While ( counter < 100 );
```

5.5 What does the following program do?

```java
public class Printing {
    public static void main( String args[] )
    {
        for ( int i = 1; i <= 10; i++ ) {
            for ( int j = 1; j <= 5; j++ )
                System.out.print( 'a' );
            System.out.println();
        }
    }
}
```

5.6 Write an application that finds the smallest of several integers. Assume that the first value read specifies the number of values to input from the user.

5.7 Write an application that calculates the product of the odd integers from 1 to 15, and then displays the results in a message dialog.

5.8 The factorial method is used frequently in probability problems. The factorial of a positive integer $n$ (written $n!$ and pronounced “$n$ factorial”) is equal to the product of the positive integers from 1 to $n$. Write an application that evaluates the factorials of the integers from 1 to 5. Display the results in tabular format in a JTextArea that is displayed on a message dialog. What difficulty might prevent you from calculating the factorial of 20?

5.9 Modify the compound-interest program of Fig. 5.6 to repeat its steps for interest rates of 5, 6, 7, 8, 9 and 10%. Use a for loop to vary the interest rate. Add scrolling functionality to the JTextArea, so you can scroll through all the output.

5.10 Write an application that displays the following patterns separately one below the other. Use for loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form `System.out.print(' '*);`. (This statement causes the asterisks to print side by side.) A statement of the form `System.out.println();` can be used to position to the next line. A statement of the form `System.out.print(' ');` can be used to display a space for the last two patterns. There should be no other output statements in the program. [Hint: The last two patterns require that each line begin with an appropriate number of blank spaces.]
5.11 One interesting application of computers is drawing graphs and bar charts (sometimes called “histograms”). Write an applet that reads five numbers, each between 1 and 30. For each number read, your program should draw a line containing that number of adjacent asterisks. For example, if your program reads the number 7, it should display "*******".

5.12 A mail-order house sells five different products whose retail prices are as follows: product 1, $2.98; product 2, $4.50; product 3, $9.98; product 4, $4.49; and product 5, $6.87. Write an application that reads a series of pairs of numbers as follows:

a) product number;
b) quantity sold for one day.

Your program should use a switch structure to help determine the retail price for each product. It should calculate and display the total retail value of all products sold last week. Use a TextField to obtain the product number from the user. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

5.13 Modify the program in Fig. 5.6 to use only integers to calculate the compound interest. [Hint: Treat all monetary amounts as integral numbers of pennies. Then “break” the result into its dollar portion and cents portion by using the division and modulus operations, respectively. Insert a period between the dollars and the cents portions.]

5.14 Assume that \( i = 1, j = 2, k = 3 \) and \( m = 2 \). What does each of the following statements print?

a) `System.out.println( i == 1 );`
b) `System.out.println( j == 3 );`
c) `System.out.println( i >= 1 && j < 4 );`
d) `System.out.println( m <= 99 & k < m );`
e) `System.out.println( j >= i || k == m );`
f) `System.out.println( k + m < j | 3 - j >= k );`
g) `System.out.println( !( k > m ) );`

5.15 Write an application that prints a table of the binary, octal, and hexadecimal equivalents of the decimal numbers in the range 1 through 256. If you are not familiar with these number systems, read Appendix E first. Place the results in a JTextArea with scrolling functionality. Display the JTextArea in a message dialog.

5.16 Calculate the value of \( \pi \) from the infinite series

\[
\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots
\]

Print a table that shows the value of \( \pi \) approximated by one term of this series, by two terms, by three terms, etc. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?
5.17  (Pythagorean Triples) A right triangle can have sides whose lengths are all integers. The set of three integer values for the lengths of the sides of a right triangle is called a Pythagorean triple. The lengths of the three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Write an application to find all Pythagorean triples for \texttt{side1}, \texttt{side2} and the \texttt{hypotenuse}, all no larger than 500. Use a triple-nested \texttt{for} loop that tries all possibilities. This method is an example of “brute force” computing. You will learn in more advanced computer science courses that there are large numbers of interesting problems for which there is no known algorithmic approach other than using sheer brute force.

5.18  Modify Exercise 5.10 to combine your code from the four separate triangles of asterisks into a single application that prints all four patterns side by side, making clever use of nested \texttt{for} loops.

5.19  (De Morgan’s Laws) In this chapter, we have discussed the logical operators \&\&, \&, \|, \|, ^ and \!. De Morgan’s Laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression \(! ( \texttt{condition1} \&\& \texttt{condition2} )\) is logically equivalent to the expression \((\! \texttt{condition1} \| \| \! \texttt{condition2} )\). Also, the expression \(! ( \texttt{condition1} \| \| \texttt{condition2} )\) is logically equivalent to the expression \((\! \texttt{condition1} \&\& \! \texttt{condition2} )\). Use De Morgan’s Laws to write equivalent expressions for each of the following, and then write a program to show that both the original expression and the new expression in each case are equivalent:

\begin{itemize}
  \item a) \(! ( x < 5 ) \&\& \! ( y >= 7 )\)
  \item b) \(! ( a == b ) \| \| \! ( g != 5 )\)
  \item c) \(! ( ( x <= 8 ) \&\& ( y > 4 ) )\)
  \item d) \(! ( ( i > 4 ) \| \| ( j <= 6 ) )\)
\end{itemize}

5.20  Write an application that prints the following diamond shape. You may use output statements that print a single asterisk (*), a single space or a single newline character. Maximize your use of repetition (with nested \texttt{for} structures,) and minimize the number of output statements.
5.21 Modify the program you wrote in Exercise 5.20 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

5.22 A criticism of the break statement and the continue statement is that each is unstructured. Actually, break statements and continue statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you would remove any break statement from a loop in a program and replace that statement with some structured equivalent. [Hint: The break statement leaves a loop from within the body of the loop. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates “early exit because of a ‘break’ condition.”] Use the technique you developed here to remove the break statement from the program in Fig. 5.11.

5.23 What does the following program segment do?

```java
for ( i = 1; i <= 5; i++ ) {
    for ( j = 1; j <= 3; j++ ) {
        for ( k = 1; k <= 4; k++ )
            System.out.print( '*' );
        System.out.println();
    }
    System.out.println();
}
```

5.24 Describe in general how you would remove any continue statement from a loop in a program and replace that statement with some structured equivalent. Use the technique you developed here to remove the continue statement from the program in Fig. 5.12.

5.25 (“The Twelve Days of Christmas” Song) Write an application that uses repetition and switch structures to print the song “The Twelve Days of Christmas.” One switch structure should be used to print the day (i.e., “First,” “Second,” etc.). A separate switch structure should be used to print the remainder of each verse. Visit the Web site www.12days.com/library/carols/12daysofxmas.htm for the complete lyrics to the song.
Methods

Objectives

• To understand how to construct programs modularly from small pieces called *methods*.
• To introduce the common math methods available in the Java API.
• To be able to create new methods.
• To understand the mechanisms for passing information between methods.
• To introduce simulation techniques that use random-number generation.
• To understand how the visibility of identifiers is limited to specific regions of programs.
• To understand how to write and use methods that call themselves.

*Form ever follows function.*
Louis Henri Sullivan

*E pluribus unum.*
*(One composed of many.)*
Virgil

*O! call back yesterday, bid time return.*
William Shakespeare, *Richard II*

*Call me Ishmael.*
Herman Melville, *Moby Dick*

*When you call me that, smile.*
Owen Wister

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
6.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters of this text. Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or modules. This technique is called divide and conquer. This chapter describes many key features of the Java language that facilitate the design, implementation, operation and maintenance of large programs.

6.2 Program Modules in Java

Modules in Java are called methods and classes. Java programs are written by combining new methods and classes the programmer writes with “prepackaged” methods and classes available in the Java API (also referred to as the Java class library) and in various other method and class libraries. In this chapter, we concentrate on methods; we begin to discuss classes in detail in Chapter 8.

The Java API provides a rich collection of classes and methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, error checking and many other useful operations. This set of modules makes the programmer’s job easier, because the modules provide many of the capabilities
programmers need. The Java API methods are provided as part of the Java 2 Software Development Kit (J2SDK).

**Good Programming Practice 6.1**

Familiarize yourself with the rich collection of classes and methods in the Java API and with the rich collections of classes available in various class libraries.

**Software Engineering Observation 6.1**

Avoid reinventing the wheel. When possible, use Java API classes and methods instead of writing new classes and methods. This reduces program development time and avoids introducing programming errors.

**Performance Tip 6.1**

Do not try to rewrite existing Java API classes and methods to make them more efficient. You usually will not be able to increase the performance of these classes and methods.

The programmer can write methods to define specific tasks that a program may use many times during its execution. These are sometimes referred to as *programmer-defined methods*. The actual statements defining the methods are written only once and are hidden from other methods.

A method is *invoked* (i.e., made to perform its designated task) by a *method call*. The method call specifies the name of the method and provides information (as *arguments*) that the called method requires to perform its task. When the method call completes, the method either returns a result to the *calling method* (or *caller*) or simply returns control to the calling method. A common analogy for this program structure is the hierarchical form of management. A boss (the calling method, or caller) asks a worker (the *called method*) to perform a task and report back (i.e., *return*) the results after completing the task. The boss method does not know how the worker method performs its designated tasks. The worker may also call other worker methods, and the boss will be unaware of this occurrence. We will soon see how this “hiding” of implementation details promotes good software engineering. Figure 6.1 shows the *boss* method communicating with several worker methods in a hierarchical manner. Note that *worker1* acts as a “boss method” to *worker4* and *worker5*. Relationships among methods may also be different than the hierarchical structure shown in this figure.

---

**Fig. 6.1** Hierarchical boss-method/worker-method relationship.
6.3 Math Class Methods

Math class methods allow the programmer to perform certain common mathematical calculations. We use various Math class methods here to introduce the concept of methods. Throughout the book, we discuss many other methods from the classes of the Java API.

Methods are called by writing the name of the method, followed by a left parenthesis, followed by the argument (or a comma-separated list of arguments) of the method, followed by a right parenthesis. For example, a programmer desiring to calculate the square root of 900.0 might write

\[ \text{Math.sqrt}(900.0) \]

When this statement executes, it calls static Math method sqrt to calculate the square root of the number contained in the parentheses (900.0). The number 900.0 is the argument of method sqrt. The preceding expression evaluates to 30.0. Method sqrt method takes an argument of type double and returns a result of type double. Note that all Math class methods are static; therefore, they are invoked by preceding the name of the method with the class name Math and a dot (.) operator. To output the value of the preceding method call in the command window, a programmer might write

\[ \text{System.out.println(Math.sqrt(900.0))}; \]

In this statement, the value that sqrt returns becomes the argument to method println.

Software Engineering Observation 6.2

It is not necessary to import class Math to use its methods. Math is part of the java.lang package, which is automatically imported by the compiler.

Common Programming Error 6.1

Forgetting to invoke a Math class method by preceding the name of the method with the class name Math and a dot operator (.) results in a syntax error.

Method arguments may be constants, variables or expressions. If \( c1 = 13.0, d = 3.0 \) and \( f = 4.0 \), then the statement

\[ \text{System.out.println(Math.sqrt(c1 + d * f))}; \]

calculates and prints the square root of \( 13.0 + 3.0 \times 4.0 = 25.0 \), namely 5.0.

Some Math class methods are summarized in Fig. 6.2. In the figure, the variables \( x \) and \( y \) are of type double. The Math class also defines two commonly used mathematical constants: Math.PI and Math.E. The constant Math.PI (3.14159265358979323846) of class Math is the ratio of a circle’s circumference to its diameter. The constant Math.E (2.7182818284590452354) is the base value for natural logarithms (calculated with static Math method log).

6.4 Methods

Methods allow the programmer to modularize a program. Variables declared in method definitions are local variables—only the method that defines them knows they exist. Most methods have a list of parameters that provide the means for communicating information between methods via method calls. A method’s parameters are also local variables.
There are several motivations for modularizing a program with methods. The divide-and-conquer approach makes program development more manageable. Another motivation is software reusability—using existing methods as building blocks to create new programs. With good method naming and definition, you can create programs from standardized methods rather than by building customized code. For example, we did not have to define how to convert Strings to integers and floating-point numbers; Java already provides such methods for us in class `Integer (parseInt)` and class `Double (parseDouble)`. A third motivation is to avoid repeating code in a program. Packaging code as a method allows a program to execute that code from several locations in a program simply by calling the method.

**Software Engineering Observation 6.3**

To promote software reusability, each method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.

---

**Table:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs( x )</code></td>
<td>absolute value of <code>x</code></td>
<td><code>abs( 23.7 )</code> is 23.7</td>
</tr>
<tr>
<td></td>
<td>(this method also has versions for <code>float</code>, <code>int</code> and <code>long</code> values)</td>
<td><code>abs( 0.0 )</code> is 0.0, <code>abs( -23.7 )</code> is 23.7</td>
</tr>
<tr>
<td><code>ceil( x )</code></td>
<td>rounds <code>x</code> to the smallest integer not less than <code>x</code></td>
<td><code>ceil( 9.2 )</code> is 10.0, <code>ceil( -9.8 )</code> is -9.0</td>
</tr>
<tr>
<td><code>cos( x )</code></td>
<td>trigonometric cosine of <code>x</code> (<code>x</code> is in radians)</td>
<td><code>cos( 0.0 )</code> is 1.0</td>
</tr>
<tr>
<td><code>exp( x )</code></td>
<td>exponential method $e^x$</td>
<td><code>exp( 1.0 )</code> is 2.71828, <code>exp( 2.0 )</code> is 7.38906</td>
</tr>
<tr>
<td><code>floor( x )</code></td>
<td>rounds <code>x</code> to the largest integer not greater than <code>x</code></td>
<td><code>floor( 9.2 )</code> is 9.0, <code>floor( -9.8 )</code> is -10.0</td>
</tr>
<tr>
<td><code>log( x )</code></td>
<td>natural logarithm of <code>x</code> (base $e$)</td>
<td><code>log( 2.71828 )</code> is 1.0, <code>log( 7.38906 )</code> is 2.0</td>
</tr>
<tr>
<td><code>max( x, y )</code></td>
<td>larger value of <code>x</code> and <code>y</code></td>
<td><code>max( 2.3, 12.7 )</code> is 12.7, <code>max( -2.3, -12.7 )</code> is -2.3</td>
</tr>
<tr>
<td></td>
<td>(this method also has versions for <code>float</code>, <code>int</code> and <code>long</code> values)</td>
<td></td>
</tr>
<tr>
<td><code>min( x, y )</code></td>
<td>smaller value of <code>x</code> and <code>y</code></td>
<td><code>min( 2.3, 12.7 )</code> is 2.3, <code>min( -2.3, -12.7 )</code> is -12.7</td>
</tr>
<tr>
<td></td>
<td>(this method also has versions for <code>float</code>, <code>int</code> and <code>long</code> values)</td>
<td></td>
</tr>
<tr>
<td><code>pow( x, y )</code></td>
<td><code>x</code> raised to power <code>y</code> ($x^y$)</td>
<td><code>pow( 2.0, 7.0 )</code> is 128.0, <code>pow( 9.0, .5 )</code> is 3.0</td>
</tr>
<tr>
<td><code>sin( x )</code></td>
<td>trigonometric sine of <code>x</code> (<code>x</code> is in radians)</td>
<td><code>sin( 0.0 )</code> is 0.0</td>
</tr>
<tr>
<td><code>sqrt( x )</code></td>
<td>square root of <code>x</code></td>
<td><code>sqrt( 900.0 )</code> is 30.0, <code>sqrt( 9.0 )</code> is 3.0</td>
</tr>
<tr>
<td><code>tan( x )</code></td>
<td>trigonometric tangent of <code>x</code> (<code>x</code> is in radians)</td>
<td><code>tan( 0.0 )</code> is 0.0</td>
</tr>
</tbody>
</table>
Software Engineering Observation 6.4

If you cannot choose a concise name that expresses a method’s task, it is possible that your method is attempting to perform too many diverse tasks. It is usually best to break such a method into several smaller method definitions.

6.5 Method Definitions

The programs presented up to this point each consisted of a class definition containing at least one method definition that called Java API methods to accomplish its tasks. We now consider how programmers write their own customized methods. Until we discuss more of the details of class definitions in Chapter 8, we use applets for all programs that contain two or more method definitions, for simplicity.

Consider an applet that uses a method \texttt{square} (invoked from the applet’s \texttt{init} method) to calculate the squares of the integers from 1 to 10 (Fig. 6.3).

When the applet begins execution, the applet container calls the applet’s \texttt{init} method. Line 16 declares \texttt{JTextArea} reference \texttt{outputArea} and initializes it with a new \texttt{JTextArea}. This \texttt{JTextArea} will display the program’s results.

This program is the first in which we display a GUI component on an applet. The on-screen display area for a \texttt{JApplet} has a \texttt{content pane}, to which the GUI components must be attached so they can be displayed at execution time. The content pane is an object of class \texttt{Container} from the \texttt{java.awt} package. This class was \texttt{imported} on line 5 for use in the applet. Line 19 declares \texttt{Container} reference \texttt{container} and assigns to it the result of a call to method \texttt{getContentPane}—one of the many methods that our class \texttt{SquareInt} inherits from class \texttt{JApplet}. Method \texttt{getContentPane} returns a reference to the applet’s content pane. The program uses that reference to attach GUI components, like a \texttt{JTextArea}, to the applet’s user interface.

Line 22 places the \texttt{JTextArea} GUI component object to which \texttt{outputArea} refers on the applet. When the applet executes, any GUI components attached to it are displayed. \texttt{Container} method \texttt{add} attaches a GUI component to a container. For the moment, we can attach only one GUI component to the applet’s content pane, and that GUI component will occupy the applet’s entire drawing area on the screen (as defined by the \texttt{width} and \texttt{height} of the applet, in pixels, in the applet’s HTML document). Later, we will discuss how to attach many GUI components to an applet by changing the applet’s \texttt{layout}. The layout controls how the applet positions GUI components in its area on the screen.

Line 24 declares \texttt{int} variable \texttt{result} to store the result of each square calculation. Line 25 declares \texttt{String} reference \texttt{output} and initializes it with the empty string. This \texttt{String} will contain the results of squaring the values from 1 to 10. Lines 28–37 define a \texttt{for} repetition structure. Each iteration of this loop calculates the \texttt{square} of the current value of control variable \texttt{x}, stores the value in \texttt{result} and concatenates the \texttt{result} to the end of \texttt{output}.

The applet invokes (or calls) its \texttt{square} method on line 31 with the statement

\begin{verbatim}
result = square( counter );
\end{verbatim}

The () after \texttt{square} represent the \texttt{method-call operator}, which has high precedence. At this point, the program makes a copy of the value of \texttt{counter} (the \texttt{argument} to the method call) and transfers program control to the first line of method \texttt{square} (defined at lines 44–
Method `square` receives the copy of the value of `counter` in the `parameter y`. Then, `square` calculates `y * y` (line 46). Method `square` uses a `return` statement to return (i.e., give back) the result of the calculation to the statement in `init` that invoked `square`. In method `init`, the return value is assigned to variable `result`. Lines 34–35 concatenate "The square of ", the value of `counter", is ", the value of `result` and a newline character to the end of `output`. This process repeats for each iteration of the `for` repetition structure. Line 39 uses method `setText` to set `outputArea`'s text to the `String` `output`.

```java
// Fig. 6.3: SquareIntegers.java
// A programmer-defined square method

// Java core packages
import java.awt.Container;

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

public class SquareIntegers extends JApplet {

   // set up GUI and calculate squares of integers from 1 to 10
   public void init() {
       // JTextArea to display results
       JTextArea outputArea = new JTextArea();

       // get applet's content pane (GUI component display area)
       Container container = getContentPane();

       // attach outputArea to container
       container.add( outputArea );

       int result; // store result of call to method square
       String output = ""; // String containing results

       // loop 10 times
       for ( int counter = 1; counter <= 10; counter++ ) {
           // calculate square of counter and store in result
           result = square( counter );

           // append result to String output
           output += "The square of " + counter + " is " + result + "\n";

       } // end for structure

       outputArea.setText( output ); // place results in JTextArea
   } // end method init
```

Fig. 6.3 Using programmer-defined method `square` (part 1 of 2).
Note that we declared references output, outputArea and container and variable result as local variables in init, because they are used only in init. Variables should be declared as instance variables only if they are required for use in more than one method of the class or if the program should save their values between calls to the class’s methods. Also, note that method init calls method square directly without preceding the method name with a class name and a dot operator or a reference name and a dot operator. Each method in a class is able to call the class’s other methods directly. However, there is an exception to this rule. A class’s static methods can call only other static methods of the class directly. Chapter 8 discusses static methods in detail.

The definition of method square (line 44) shows that square expects an integer parameter y; square uses this name to manipulate the value it receives. Keyword int preceding the name of the method indicates that square returns an integer result. The return statement in square passes the result of the calculation y * y back to the calling method. Note that the entire method definition appears between the braces of the class SquareInt. All methods must be defined inside a class definition.

**Good Programming Practice 6.2**

Place a blank line between method definitions to separate the methods and enhance program readability.

**Common Programming Error 6.2**

Defining a method outside the braces of a class definition is a syntax error.
The general format of a method definition is

\[
\text{return-value-type \ method-name( parameter-list )} \\
\{ \\
\text{declarations and statements} \\
\}
\]

The method-name is any valid identifier. The return-value-type is the data type of the result returned from the method to the caller. The return-value-type void indicates that a method does not return a value. Methods can return at most one value.

The parameter-list is a comma-separated list in which the method declares each parameter’s type and name. There must be one argument in the method call for each parameter in the method definition. Each argument also must be compatible with the type of the corresponding parameter in the method definition. For example, a parameter of type double can receive values of 7.35, 22 or –0.03546, but not "hello" (because a String cannot be assigned to a double variable). If a method does not receive any values, the parameter-list is empty (i.e., the name of the method is followed by an empty set of parentheses). Each parameter in the parameter list of a method must be declared with a data type; otherwise, a syntax error occurs.

Following the first line of the method definition (also known as the method header), declarations and statements in braces form the method body. The method body is also referred to as a block. Variables can be declared in any block, and blocks can be nested. A method cannot be defined inside another method.

There are three ways to return control to the statement that invoked a method. If the method does not return a result, control returns when the program flow reaches the method-ending right brace or when the statement

\[\text{return;}\]

is executed. If the method returns a result, the statement

\[\text{return \ expression;}\]

evaluates the expression, then returns the resulting value to the caller. When a return statement executes, control returns immediately to the statement that invoked the method.

Note that the example in Fig. 6.3 actually contains two method definitions—init (lines 13–41) and square (line 44–48). Remember that the applet container calls method init to initialize the applet. In this example, method init repeatedly invokes the square method to perform a calculation, then places the results in the JTextArea that is attached to the applet’s content pane. When the applet appears on the screen, the results are displayed in the JTextArea.

Notice the syntax used to invoke method square—we use just the name of the method, followed by the arguments to the method in parentheses. Methods in a class definition are allowed to invoke other methods in the same class definition by using this syntax. (There is an exception to this rule, discussed in Chapter 8.) Methods in the same class definition are both the methods defined in that class and the inherited methods (the methods from the class that the current class extends—JApplet in Fig. 6.3). We have now seen three ways to call a method: A method name by itself (as shown with square(x) in this example), a reference to an object followed by the dot (.) operator and the method name.
(such as `g.drawLine( x1, y1, x2, y2 )`), and a class name followed by a method name (such as `Integer.parseInt( stringToConvert )`). The last syntax is only for `static` methods of a class (discussed in detail in Chapter 8).

**Common Programming Error 6.3**
Omitting the return-value-type in a method definition is a syntax error.

**Common Programming Error 6.4**
Forgetting to return a value from a method that should return a value is a syntax error. If a return value type other than `void` is specified, the method must contain a `return` statement.

**Common Programming Error 6.5**
Returning a value from a method whose return type has been declared `void` is a syntax error.

**Common Programming Error 6.6**
Declaring method parameters of the same type as `float x, y` instead of `float x, float y` is a syntax error, because types are required for each parameter in the parameter list.

**Common Programming Error 6.7**
Placing a semicolon after the right parenthesis enclosing the parameter list of a method definition is a syntax error.

**Common Programming Error 6.8**
Redefining a method parameter in the method’s body is a syntax error.

**Common Programming Error 6.9**
Passing to a method an argument that is not compatible with the corresponding parameter’s type is a syntax error.

**Common Programming Error 6.10**
Defining a method inside another method is a syntax error.

**Good Programming Practice 6.3**
Avoid using the same names for instance variables and local variables. This helps readers of your program distinguish variables used in different parts of a class definition.

**Good Programming Practice 6.4**
Choosing meaningful method names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.

**Software Engineering Observation 6.5**
A method should usually be no longer than one page. Better yet, a method should usually be no longer than half a page. Regardless of how long a method is, it should perform one task well. Small methods promote software reusability.

**Software Engineering Observation 6.6**
Programs should be written as collections of small methods. This makes programs easier to write, debug, maintain and modify.
A method requiring a large number of parameters may be performing too many tasks. Consider dividing the method into smaller methods that perform the separate tasks. The method header should fit on one line if possible.

The method header and method calls must all agree in the number, type and order of parameters and arguments.

Small methods are easier to test, debug and understand than large ones.

The applet in our next example (Fig. 6.4) uses a programmer-defined method called `maximum` to determine and return the largest of three floating-point values.

```java
// Fig. 6.4: Maximum.java
// Finding the maximum of three doubles

import java.awt.Container;
import javax.swing.*;

public class Maximum extends JApplet {

    public void init() {
        // obtain user input
        String s1 = JOptionPane.showInputDialog("Enter first floating-point value");
        String s2 = JOptionPane.showInputDialog("Enter second floating-point value");
        String s3 = JOptionPane.showInputDialog("Enter third floating-point value");

        // convert user input to double values
        double number1 = Double.parseDouble(s1);
        double number2 = Double.parseDouble(s2);
        double number3 = Double.parseDouble(s3);

        // call method maximum to determine largest value
        double max = maximum(number1, number2, number3);

        // create JTextArea to display results
        JTextArea outputArea = new JTextArea();
    }

    public static double maximum(double number1, double number2, double number3) {
        if (number1 > number2 && number1 > number3) return number1;
        if (number2 > number1 && number2 > number3) return number2;
        return number3;
    }
}
```

**Fig. 6.4** Programmer-defined `maximum` method (part 1 of 2).
Chapter 6 Methods

The three floating-point values are input by the user via input dialogs (lines 16–21 of `init`). Lines 24–26 use method `Double.parseDouble` to convert the `String` input by the user to `double` values. Line 29 calls method `maximum` (defined on lines 49–53) to determine the largest `double` value of the three `double` values passed as arguments to the method. Method `maximum` returns the result to method `init`, using a `return` statement. The program assigns the result to variable `max`. Lines 35–37 use `String` concatenation to form a `String` containing the three `double` values input by the user and the `max` value and place the result in `JTextArea outputArea`.

Notice the implementation of the method `maximum` (lines 49–53). The first line indicates that the method returns a `double` floating-point value, that the method’s name is `maximum` and that the method takes three `double` parameters (`x`, `y` and `z`) to accomplish its task. Also, the statement (line 51) in the body of the method returns the largest of the
three floating-point values, using two calls to the `Math.max` method. First, the statement invokes method `Math.max` with the values of variables `y` and `z` to determine the larger of the two values. Next, the statement passes the value of variable `x` and the result of the first call to `Math.max` to method `Math.max`. Finally, the statement returns the result of the second call to `Math.max` to line 29 (the point at which method `init` invoked method `maximum`).

### 6.6 Argument Promotion

Another important feature of method definitions is the coercion of arguments (i.e., the forcing of arguments to the appropriate type to pass to a method). For example, a program can call `Math` method `sqrt` with an integer argument even though the method expects to receive a `double` argument. For example, the statement

```java
System.out.println( Math.sqrt( 4 ) );
```

correctly evaluates `Math.sqrt(4)` and prints the value `2`. The method definition’s parameter list causes Java to convert the integer value `4` to the `double` value `4.0` before passing the value to `sqrt`. In some cases, attempting these conversions leads to compiler errors if Java’s promotion rules are not satisfied. The promotion rules specify how to convert types to other types without losing data. In our `sqrt` example above, an `int` is converted to a `double` without changing its value. However, converting a `double` to an `int` truncates the fractional part of the `double` value. Converting large integer types to small integer types (e.g., `long` to `int`) may also result in changed values.

The promotion rules apply to expressions containing values of two or more data types (also referred to as mixed-type expressions) and to primitive-data-type values passed as arguments to methods. The type of each value in a mixed-type expression is promoted to the “highest” type in the expression (actually, the expression uses a temporary copy of each value; the original values remain unchanged). The type of a method argument can be promoted to any “higher” type. Figure 6.5 lists the primitive data types and the types to which each is allowed to be promoted automatically.

<table>
<thead>
<tr>
<th>Type</th>
<th>Allowed promotions</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>None</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>long</td>
<td>float or double</td>
</tr>
<tr>
<td>int</td>
<td>long, float or double</td>
</tr>
<tr>
<td>char</td>
<td>int, long, float or double</td>
</tr>
<tr>
<td>short</td>
<td>int, long, float or double</td>
</tr>
<tr>
<td>byte</td>
<td>short, int, long, float or double</td>
</tr>
<tr>
<td>boolean</td>
<td>None (boolean values are not considered to be numbers in Java)</td>
</tr>
</tbody>
</table>

**Fig. 6.5** Allowed promotions for primitive data types.
Converting values to lower types can result in different values. Therefore, in cases where information may be lost due to conversion, the Java compiler requires the programmer to use a cast operator to force the conversion to occur. To invoke our `square` method, which uses an integer parameter with the `double` variable `y` (Fig. 6.3), we write the method call as `square((int)y)`. This method call explicitly casts (converts) the value of `y` to an integer for use in method `square`. Thus, if `y`’s value is `4.5`, method `square` returns `16`, not `20.25`.

**Common Programming Error 6.11**

Converting a primitive-data-type value to another primitive data type may change the value if the new data type is not an allowed promotion (e.g., `double` to `int`). Also, converting any integral value to a floating-point value and back to an integral value may introduce rounding errors into the result.

### 6.7 Java API Packages

As we have seen, Java contains many predefined classes that are grouped into categories of related classes, called packages. Together, we refer to these packages as the *Java applications programming interface* (*Java API*), or the *Java class library*.

Throughout the text, `import` statements specify the classes required to compile a Java program. For example, a program uses the statement

```java
import javax.swing.JApplet;
```

to tell the compiler to load the `JApplet` class from the `javax.swing` package. One of the great strengths of Java is the large number of classes in the packages of the Java API that programmers can reuse rather than “reinventing the wheel.” We exercise a large number of these classes in this book. Figure 6.6 lists a subset of the many packages in the Java API and provides a brief description of each package. We use classes from these packages and others throughout this book. We provide this table to begin introducing you the variety of reusable components available in the Java API. When learning Java, you should spend time reading the descriptions of the packages and classes in the Java API documentation ([java.sun.com/j2se/1.3/docs/api](http://java.sun.com/j2se/1.3/docs/api)).

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.applet</td>
<td><em>The Java Applet Package.</em>&lt;br&gt;This package contains the <code>Applet</code> class and several interfaces that enable the creation of applets, interaction of applets with the browser and playing audio clips. In Java 2, class <code>javax.swing.JApplet</code> is used to define an applet that uses the <em>Swing GUI components</em>.</td>
</tr>
<tr>
<td>java.awt</td>
<td><em>The Java Abstract Windowing Toolkit Package.</em>&lt;br&gt;This package contains the classes and interfaces required to create and manipulate graphical user interfaces in Java 1.0 and 1.1. In Java 2, these classes can still be used, but the <em>Swing GUI components</em> of the <code>javax.swing</code> packages are often used instead.</td>
</tr>
</tbody>
</table>
The set of packages available in the Java 2 Software Development Kit (J2SDK) is quite large. In addition to the packages summarized in Fig. 6.6, the J2SDK includes packages for complex graphics, advanced graphical user interfaces, printing, advanced networking, security, database processing, multimedia, accessibility (for people with disabilities) and many other functions. For an overview of the packages in the J2SDK version 1.3, visit java.sun.com/j2se/1.3/docs/api/overview-summary.html

Also, many other packages are available for download at java.sun.com.
6.8 Random-Number Generation

We now take a brief and, hopefully, entertaining diversion into a popular programming application, namely, simulation and game playing. In this section and the next section, we will develop a nicely structured game-playing program that includes multiple methods. The program uses most of the control structures we have studied to this point in the book and introduces several new concepts.

There is something in the air of a gambling casino that invigorates people—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It is the element of chance, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced through the `random` method from the `Math` class. [Note: Java also provides a `Random` class in package `java.util`. Class `Random` is covered in Chapter 20.]

Consider the following statement:

```java
double randomNumber = Math.random();
```

The `random` method of class `Math` generates a random `double` value from 0.0 up to, but not including, 1.0. If method `random` truly produces values at random, then every value from 0.0 up to, but not including, 1.0 should have an equal chance (or probability) of being chosen each time method `random` is called. Note that the values returned by `random` are actually pseudo-random numbers—a sequence of values produced by a complex mathematical calculation. That calculation uses the current time of day to seed the random number generator, such that each execution of a program yields a different sequence of random values.

The range of values produced directly by method `random` often is different from the range of values required in a particular Java application. For example, a program that simulates coin tossing might require only 0 for “heads” and 1 for “tails.” A program that simulates rolling a six-sided die would require random integers in the range from 1 to 6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game would require random integers in the range from 1 to 4.

To demonstrate method `random`, let us develop a program that simulates 20 rolls of a six-sided die and displays the value of each roll. We use the multiplication operator (*) in conjunction with method `random` as follows to produce integers in the range from 0 to 5:

```java
(int) ( Math.random() * 6 )
```

This manipulation is called scaling the range of values produced by `Math` method `random`. The number 6 in the preceding expression is called the scaling factor. The integer cast operator truncates the floating-point part (the part after the decimal point) of each value produced by the expression. Next, we shift the range of numbers produced by adding 1 to our previous result, as in

```java
1 + (int) ( Math.random() * 6 )
```

Figure 6.7 confirms that the results of the preceding calculation are integers in the range from 1 to 6.
Fig. 6.7: Shifted and scaled random integers.
To show that these numbers occur with approximately equal likelihood, let us simulate 6000 rolls of a die with the program in Fig. 6.8. Each integer from 1 to 6 should appear approximately 1000 times.

```java
// Fig. 6.8: RollDie.java
// Roll a six-sided die 6000 times.

import javax.swing.*;

public class RollDie {
    public static void main( String args[] )
    {
        int frequency1 = 0, frequency2 = 0, frequency3 = 0,
            frequency4 = 0, frequency5 = 0, frequency6 = 0, face;

        // summarize results
        for ( int roll = 1; roll <= 6000; roll++ ) {
            face = 1 + ( int ) ( Math.random() * 6 );

            // determine roll value and increment appropriate counter
            switch ( face ) {
                case 1:
                    ++frequency1;
                    break;

                case 2:
                    ++frequency2;
                    break;

                case 3:
                    ++frequency3;
                    break;

                case 4:
                    ++frequency4;
                    break;

                case 5:
                    ++frequency5;
                    break;

                case 6:
                    ++frequency6;
                    break;
            }  // end switch structure
        }  // end for structure
    }  // end main method
}
```

Fig. 6.8 Rolling a six-sided die 6000 times (part 1 of 2).
As the program output shows, scaling and shifting the values produced by method `random` enables the program to simulate realistically the rolling of a six-sided die. Note that the use of nested control structures in the program to determine the number of times each side of the six-sided die occurred. The `for` structure at lines 16–48 iterates 6000 times. During each iteration of the loop, line 17 produces a value from 1 to 6. The nested `switch` structure at lines 20–46 uses as its controlling expression the `face` value that was randomly chosen. Based on the value of `face`, the `switch` structure increments one of the six counter variables during each iteration of the loop. Note that the `switch` structure has no `default` case. When we study arrays in Chapter 7, we show how to replace the entire `switch` structure in this program with a single statement. Run the program several times, and observe the results. Notice that the program produces different results each time the program executes.

The values produced directly by `random` are always in the range

\[ 0.0 \leq \text{Math.random()} < 1.0 \]

Previously, we demonstrated how to write a single statement to simulate the rolling of a six-sided die with the statement

\[
\text{face} = 1 + (\text{int}) (\text{Math.random()} * 6);
\]
which always assigns an integer (at random) to variable face in the range 1 ≤ face ≤ 6. Note that the width of this range (i.e., the number of consecutive integers in the range) is 6, and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale random with the multiplication operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) added to (int) (Math.random() * 6). We can generalize this result as

\[ n = a + \text{(int)} (\text{Math.random()} \times b); \]

where a is the shifting value (which is equal to the first number in the desired range of consecutive integers) and b is the scaling factor (which is equal to the width of the desired range of consecutive integers). In the exercises, we will see that it is possible to choose integers at random from sets of values other than ranges of consecutive integers.

### 6.9 Example: A Game of Chance

One of the most popular games of chance is a dice game known as “craps,” which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> A player rolls two dice. Each die has six faces. These faces contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll your point value). The player loses by rolling a 7 before making the point.

The applet in Fig. 6.9 simulates the game of craps.
// graphical user interface components
JLabel die1Label, die2Label, sumLabel, pointLabel;
JTextField die1Field, die2Field, sumField, pointField;
JButton rollButton;

// set up GUI components
public void init()
{
    // obtain content pane and change its layout to
    // a FlowLayout
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    // create label and text field for die 1
    die1Label = new JLabel( "Die 1" );
    container.add( die1Label );
    die1Field = new JTextField( 10 );
    die1Field.setEditable( false );
    container.add( die1Field );

    // create label and text field for die 2
    die2Label = new JLabel( "Die 2" );
    container.add( die2Label );
    die2Field = new JTextField( 10 );
    die2Field.setEditable( false );
    container.add( die2Field );

    // create label and text field for sum
    sumLabel = new JLabel( "Sum is" );
    container.add( sumLabel );
    sumField = new JTextField( 10 );
    sumField.setEditable( false );
    container.add( sumField );

    // create label and text field for point
    pointLabel = new JLabel( "Point is" );
    container.add( pointLabel );
    pointField = new JTextField( 10 );
    pointField.setEditable( false );
    container.add( pointField );

    // create button user clicks to roll dice
    rollButton = new JButton( "Roll Dice" );
    rollButton.addActionListener( this );
    container.add( rollButton );
}

// process one roll of dice
public void actionPerformed( ActionEvent actionEvent )
{
    // first roll of dice
    if ( firstRoll ) {
        sumOfDice = rollDice();     // roll dice
switch (sumOfDice) {
    // win on first roll
    case 7: case 11:
        gameStatus = WON;
        pointField.setText("" ); // clear point field
        break;
    // lose on first roll
    case 2: case 3: case 12:
        gameStatus = LOST;
        pointField.setText("" ); // clear point field
        break;
    // remember point
    default:
        gameStatus = CONTINUE;
        myPoint = sumOfDice;
        pointField.setText( Integer.toString( myPoint ) );
        firstRoll = false;
        break;
} // end switch structure

} // end if structure body

// subsequent roll of dice
else {
    sumOfDice = rollDice(); // roll dice
    // determine game status
    if (sumOfDice == myPoint) // win by making point
        gameStatus = WON;
    else
        if (sumOfDice == 7) // lose by rolling 7
            gameStatus = LOST;
    }
    // display message indicating game status
    displayMessage();
} // end method actionPerformed

// roll dice, calculate sum and display results
public int rollDice()
{
    int die1, die2, sum;

    // pick random die values
    die1 = 1 + ( int ) ( Math.random() * 6 );
    die2 = 1 + ( int ) ( Math.random() * 6 );

Fig. 6.9 Program to simulate the game of craps (part 3 of 5).
```java
128    sum = die1 + die2;   // sum die values
129
130    // display results
131    die1Field.setText( Integer.toString( die1 ) );
132    die2Field.setText( Integer.toString( die2 ) );
133    sumField.setText( Integer.toString( sum ) );
134
135    return sum;   // return sum of dice
136
137 }  // end method rollDice
138
139 // determine game status and display appropriate message
140 // in status bar
141 public void displayMessage()
142 {
143    // game should continue
144    if ( gameStatus == CONTINUE )
145        showStatus( "Roll again." );
146
147    // game won or lost
148    else {
149
150        if ( gameStatus == WON )
151            showStatus( "Player wins. " +
152                          "Click Roll Dice to play again." );
153        else
154            showStatus( "Player loses. " +
155                          "Click Roll Dice to play again." );
156
157        // next roll is first roll of new game
158        firstRoll = true;
159    }
160
161 }  // end method displayMessage
162
163 }  // end class Craps
```

A **JLabel** object  
A **JTextField** object  
A **JButton** object

Fig. 6.9 Program to simulate the game of craps (part 4 of 5).
Notice that the player must roll two dice on the first and all subsequent rolls. When you execute the applet, click the **Roll Dice** button to play the game. The lower left corner of the **appletviewer** window displays the result of each roll. The screen captures show four separate executions of the applet (a win and a loss on the first roll, and a win and a loss after the first roll).

Until now, most user interactions in our programs have been through either an input dialog (in which the user could type an input value for the program) or a message dialog (in which a message was displayed to the user, and the user could click **OK** to dismiss the dialog). Although these dialogs are valid ways to receive input from a user and display output in a Java program, their capabilities are fairly limited—an input dialog can obtain only one value at a time from the user and a message dialog can display only one message. It is much more common to receive multiple inputs from the user at once (such as the user entering name and address information) or display many pieces of data at once (such as the values of the dice, the sum of the dice and the point, in this example). To begin our introduction to more elaborate user interfaces, this program illustrates two new graphical user interface concepts: Attaching several GUI components to an applet and **GUI event handling**. We discuss each of the new issues as they are encountered in the program.

The **import** statements in lines 5–9 enable the compiler to load the classes used in this applet. Line 5 specifies that the program uses classes from package **java.awt** (specifically, classes **Container** and **FlowLayout**). Line 6 specifies that the program uses classes from package **java.awt.event**. This package contains many data types that enable a program to process a user’s interactions with a program’s GUI. In this program, we use the **ActionListener** and **ActionEvent** data types from package **java.awt.event**. Line 9 specifies that the program uses classes from package **javax.swing** (specifically, **JApplet**, **JLabel**, **JTextField** and **JButton**).

As stated earlier, every Java program is based on at least one class definition that extends and enhances an existing class definition via inheritance. Remember that applets inherit from class **JApplet**. Line 11 indicates that class **Craps** inherits from **JApplet**.
and implements ActionListener. A class can inherit existing attributes and behaviors (data and methods) from another class specified to the right of keyword extends in the class definition. In addition, a class can implement one or more interfaces. An interface specifies one or more behaviors (i.e., methods), which you must define in your class definition. Implementing interface ActionListener forces us to define a method with the first line

    public void actionPerformed( ActionEvent actionEvent )

in our Craps class. This method’s task is to process a user’s interaction with the JButton (called Roll Dice on the user interface). When the user presses the button, this method will be called automatically in response to the user interaction. This process is called event handling. The event is the user interaction (i.e., pressing the button). The event handler is method actionPerformed. We discuss the details of this interaction and method actionPerformed shortly. Chapter 9, Object-Oriented Programming, discusses interfaces in detail. For now, as you develop your own applets that have graphical user interfaces, mimic the features that support event handling of the GUI components we present.

The game of craps is reasonably involved. The player may win or lose on the first roll, or may win or lose on any roll. Line 14 creates variables that define the three states of a game of craps: Game won (WON), game lost (LOST) or continue rolling the dice (CONTINUE). Keyword final at the beginning of the declaration indicates that these are constant variables. When a program declares a final variable, the program must initialize the variable before using the variable and cannot modify the variable thereafter. If the variable is an instance variable, this initialization normally occurs in the variable’s declaration. The initialization also can occur in a special method of a class called a constructor (discussed in Chapter 8). Constant variables are often called named constants or read-only variables. We provide more details on keyword final in Chapter 7 and Chapter 8.

**Common Programming Error 6.12**

After declaring and initializing a final variable, attempting to assign another value to that variable is a syntax error.

**Good Programming Practice 6.5**

Use only uppercase letters (with underscores between words) in the names of final variables. This format makes these constants stand out in a program.

**Good Programming Practice 6.6**

Using meaningfully named final variables rather than integer constants (such as 2) makes programs more readable.

Lines 17–20 declare several instance variables that are used throughout the Craps applet. Variable firstRoll is a boolean variable that indicates whether the next roll of the dice is the first roll in the current game. Variable sumOfDice maintains the sum of the dice for the last roll. Variable myPoint stores the “point” if the player does not win or lose on the first roll. Variable gameStatus keeps track of the current state of the game (WON, LOST or CONTINUE).

Lines 23–25 declare references to the GUI components used in this applet’s graphical user interface. References die1Label, die2Label, sumLabel and pointLabel all refer to JLabel objects. A JLabel contains a string of characters to be displayed on the
screen. Normally, a JLabel indicates the purpose of another GUI component on the screen. In the screen captures of Fig. 6.9, the JLabel objects are the text to the left of each rectangle in the first two rows of the user interface. References die1Field, die2Field, sumField and pointField all refer to JTextField objects. JTextField objects are used to get a single line of information from the user at the keyboard or to display information on the screen. The JTextField objects are the rectangles to the right of each JLabel in the first two rows of the user interface. Reference rollButton refers to a JButton object. When the user presses a JButton, the program normally responds by performing a task (rolling the dice, in this example). The JButton object is the rectangle containing the words Roll Dice at the bottom of the user interface shown in Fig. 6.9.

We have seen JButtons in prior programs—every message dialog and every input dialog contained an OK button to dismiss the message dialog or send the user’s input to the program. We also have seen JTextFields in prior programs—every input dialog contains a JTextField in which the user types an input value.

Method init (lines 28–67) creates the GUI component objects and attaches them to the user interface. Line 32 declares Container reference container and assigns to it the result method getContentPane. Remember, method getContentPane returns a reference to the applet’s content pane that can be used to attach GUI components to the applet’s user interface.

Line 33 uses Container method setLayout to specify the layout manager for the applet’s user interface. Layout managers arrange GUI components on a Container for presentation purposes. The layout managers determine the position and size of every GUI component attached to the container, thereby processing most of the layout details and enabling the programmer to concentrate on the basic look and feel of the programs.

FlowLayout is the simplest layout manager. GUI components are placed from left to right in the order in which they are attached to the Container (the applet’s content pane in this example) with method add. When the layout manager reaches the edge of the container, it begins a new row of components and continues laying out the components on that row. Line 33 creates a new object of class FlowLayout and passes it as the argument to method setLayout. Normally, the layout is set before any GUI components are added to a Container.

Common Programming Error 6.13

If a Container is not large enough to display the GUI component attached to it, some or all of the GUI components simply will not display.

[Note: Each Container can have only one layout manager at a time. Separate Containers in the same program can have different layout managers. Most Java programming environments provide GUI design tools that help a programmer graphically design a GUI; then the tools write Java code to create the GUI. Some of these GUI design tools also allow the programmer to use layout managers. Chapter 12 and Chapter 13 discuss several layout managers that allow more precise control over the layout of the GUI components.]

Lines 36–40, 43–47, 50–54 and 57–61 each create a JLabel and JTextField pair and attach them to the user interface. Since these sets of lines are all quite similar, we concentrate on lines 36–40. Line 36 creates a new JLabel object, initializes it with the string "Die 1" and assigns the object to reference die1Label. This procedure labels the corresponding JTextField (named die1Field) in the user interface, so the user can deter-
mine the purpose of the value displayed in die1Field. Line 37 attaches the JLabel to which die1Label refers to the applet’s content pane. Line 38 creates a new JTextField object, initializes it to be 10 characters wide and assigns the object to reference die1Field. This JTextField displays the value of the first die after each roll of the dice. Line 39 uses JTextField method setEditable with the argument false to indicate that the user should not be able to type in the JTextField. This setting makes the JTextField uneditable and causes it to be displayed with a gray background by default. An editable JTextField has a white background (as seen in input dialogs). Line 32 attaches the JTextField to which die1Field refers to the applet’s content pane.

Line 64 creates a new JButton object, initializes it with the string "Roll Dice" (this string will appear on the button) and assigns the object to reference rollButton.

Line 65 specifies that this applet should listen for events from the rollButton. The this keyword enables the applet to refer to itself. (We discuss this in detail in Chapter 8.) When the user interacts with a GUI component an event is sent to the applet. GUI events are messages (method calls) indicating that the user of the program has interacted with one of the program’s GUI components. For example, when you press rollButton in this program, a message indicating the event that occurred is sent to the applet to notify the applet that you pressed the button. For a JButton, the message indicates to the applet that an action was performed by the user on the JButton and automatically calls method actionPerformed to process the user’s interaction.

This style of programming is known as event-driven programming—the user interacts with a GUI component, the program is notified of the event and the program processes the event. The user’s interaction with the GUI “drives” the program. The methods that are called when an event occurs are also known as event-handling methods. When a GUI event occurs in a program, Java creates an object containing information about the event that occurred and calls an appropriate event-handling method. Before any event can be processed, each GUI component must know which object in the program defines the event-handling method that will be called when an event occurs. In line 65, JButton method addActionListener is used to tell rollButton that the applet (this) can listen for action events and defines method actionPerformed. This procedure is called registering the event handler with the GUI component. (We also like to call it the start-listening line, because the applet is now listening for events from the button.) To respond to an action event, we must define a class that implements ActionListener (this requires that the class also define method actionPerformed), and we must register the event handler with the GUI component. Finally, the last line in init attaches the JButton to which roll refers to the applet’s content pane, thus completing the user interface.

Method actionPerformed (lines 70–117) is one of several methods that process interactions between the user and GUI components. The first line of the method indicates that actionPerformed is a public method that returns nothing (void) when it completes its task. Method actionPerformed receives one argument—an ActionEvent—when it is called in response to an action performed on a GUI component by the user (in this case, pressing the JButton). The ActionEvent argument contains information about the action that occurred.

We define method rollDice (lines 120–137) to roll the dice and compute and display their sum. Method rollDice is defined once, but it is called from two places in the program (lines 74 and 104). Method rollDice takes no arguments, so it has an empty
parameter list. Method `rollDice` returns the sum of the two dice, so a return type of `int` is indicated in the method’s header.

The user clicks `Roll Dice` to roll the dice. This action invokes method `actionPerformed` (line 70) of the applet. Method `actionPerformed` checks the boolean variable `firstRoll` (line 73) to determine if it is `true` or `false`. If it is `true`, this roll is the first roll of the game. Line 74 calls `rollDice`, which picks two random values from 1 to 6, displays the value of the first die, second die and the sum of the dice in the first three `JTextField`s, respectively, and returns the sum of the dice. Note that the integer values are converted to `Strings` (lines 131–133) with `static` method `Integer.toString`, because `JTextField`s can display only `Strings`. After the first roll, the nested `switch` structure at line 76 in `actionPerformed` determines if the game has been won or lost, or if the game should continue with another roll. After the first roll, if the game is not over, `sumOfDice` is saved in `myPoint` and displayed in `pointField`.

Line 115 calls method `displayMessage` (defined at lines 141–161) to display the current status of the game. The `if/else` structure at line 144 uses applet method `showStatus` to display a `String` in the applet container’s status bar. Line 145 displays

```
Roll again.
```

if `gameStatus` is equal to `CONTINUE`. Lines 150–151 display

```
Player wins. Click Roll Dice to play again.
```

if `gameStatus` is equal to `WON`. Lines 153–154 display

```
Player loses. Click Roll Dice to play again.
```

if `gameStatus` is equal to `LOST`. Method `showStatus` receives a `String` argument and displays it in the status bar of the applet container. If the game is over (i.e., it has been won or lost) line 158 sets `firstRoll` to `true` to indicate that the next roll of the dice is the first roll of the next game.

The program then waits for the user to click the `Roll Dice` button again. Each time the user presses `Roll Dice` button, method `actionPerformed` invokes method `rollDice` to produce a new `sumOfDice`. If the current roll is a continuation of an incomplete game, the code in lines 103–112 executes. In line 107, if `sumOfDice` matches `myPoint`, line 108 sets `gameStatus` to `WON`, and the game is complete. In line 110, if `sumOfDice` is equal to 7, line 111 sets `gameStatus` to `LOST`, and the game is complete. When the game completes, `displayMessage` displays an appropriate message, and the user can click the `Roll Dice` button to begin a new game. Throughout the program, the four `JTextField`s are updated with the new values of the dice and the sum on each roll, and the `pointField` is updated each time a new game begins.

Note the interesting use of the various program control mechanisms we have discussed. The craps program uses four methods—`init`, `actionPerformed`, `rollDice` and `displayMessage`—and the `switch`, `if/else` and nested `if` structures. Note also the use of multiple `case` labels in the `switch` structure to execute the same statements (lines 79 and 85). Also, note that the event-handling mechanism acts as a form of program control. In this program, event-handling enables user-controlled repetition—each time the user clicks `Roll Dice`, the program rolls the dice again. In the exercises, we investigate various interesting characteristics of the game of craps.
6.10 Duration of Identifiers

Chapter 2 through Chapter 5 used identifiers for variable names and reference names. The attributes of variables and references include name, type, size and value. We also use identifiers as names for user-defined methods and classes. Actually, each identifier in a program has other attributes, including duration and scope.

An identifier’s duration (also called its lifetime) is the period during which the identifier exists in memory. Some identifiers exist for brief periods of time and others exist for the entire execution of a program.

An identifier’s scope defines where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program, while others can be referenced only from limited portions of a program. This section discusses duration of identifiers. Section 6.11 discusses the scope of identifiers.

Identifiers that represent local variables in a method (i.e., parameters and variables declared in the body of the method) have automatic duration. Automatic-duration variables are created when program control reaches their declaration; they exist while the block in which they are declared is active; and they are destroyed when the block in which they are declared is exited. We will continue to refer to variables of automatic duration as local variables.

Performance Tip 6.2

Automatic duration is a means of conserving memory, because automatic-duration variables are created when program control reaches their declaration and are destroyed when the block in which they are declared is exited.

The instance variables of a class are initialized automatically by the compiler if the programmer does not provide explicit initial values. Variables of the primitive data types are initialized to zero, except boolean variables, which are initialized to false. References are initialized to null. Unlike instance variables of a class, automatic variables must be initialized by the programmer before they can be used.

Testing and Debugging Tip 6.2

If an automatic variable is not initialized before it is used in a method, the compiler issues an error message.

Java also has identifiers of static duration. Variables and references of static duration exist from the point at which the class that defines them is loaded into memory for execution until the program terminates. Their storage is allocated and initialized when their classes are loaded into memory. Even though static-duration variables and reference names exist when their classes are loaded into memory, these identifiers cannot necessarily be used throughout a program. Duration (an identifier’s lifetime) and scope (where an identifier can be used) are separate issues, as shown in Section 6.11.

Software Engineering Observation 6.9

Automatic duration is an example of the principle of least privilege. This principle states that each component of a system should have sufficient rights and privileges to accomplish its designated task, but no additional rights or privileges. This constraint helps prevent accidental and/or malicious errors from occurring in systems. Why have variables stored in memory and accessible when they are not needed?
6.11 Scope Rules

The *scope* of an identifier for a variable, reference or method is the portion of the program that can reference the identifier. A local variable or reference declared in a block can be used only in that block or in blocks nested within that block. The scopes for an identifier are *class scope* and *block scope*. There is also a special scope for labels used with the `break` and `continue` statements (introduced in Chapter 5, “Control Structures: Part 2”). A label is visible only in the body of the repetition structure that immediately follows the label.

Methods and instance variables of a class have *class scope*. Class scope begins at the opening left brace, `{`, of the class definition and terminates at the closing right brace, `}`, of the class definition. Class scope enables methods of a class to invoke directly all methods defined in that same class or inherited into that class (such as the methods inherited into our applets from class `JApplet`) and to access directly all instance variables defined in the class. In Chapter 8, we will see that static methods are an exception to this rule. In a sense, all instance variables and methods of a class are *global* to the methods of the class in which they are defined (i.e., the methods can modify the instance variables directly and invoke other methods of the class). [Note: One of the reasons we use mainly applets in this chapter is to simplify our discussions. We have not as yet introduced a true windowed application in which the methods of our application class will have access to all the other methods of the class and the instance variables of the class.]

Identifiers declared inside a block have *block scope*. Block scope begins at the identifier’s declaration and ends at the terminating right brace (`}`) of the block. Local variables of a method have block scope, as do method parameters, which are also local variables of the method. Any block may contain variable or reference declarations. When blocks are nested in a method’s body and an identifier declared in an outer block has the same name as an identifier declared in an inner block, the compiler generates a syntax error stating that the variable is already defined. If a local variable in a method has the same name as an instance variable, the instance variable is “hidden” until the block terminates execution. In Chapter 8, we discuss how to access such “hidden” instance variables.

**Common Programming Error 6.14**

Accidentally using the same name for an identifier in an inner block of a method as is used for an identifier in an outer block of the same method results in a syntax error from the compiler.

**Good Programming Practice 6.7**

Avoid local-variable names that hide instance-variable names. This can be accomplished by avoiding the use of duplicate identifiers in a class.

The applet of Fig. 6.10 demonstrates scoping issues with instance variables and local variables.

This example uses the applet’s `start` method (lines 27–40) for the first time. Remember, when an applet container loads an applet, the container first creates an instance of the applet class. It then calls the applet’s `init`, `start` and `paint` methods. Method `start` always is defined with the line shown on line 27 as its first line.
public class Scoping extends JApplet {
    JTextArea outputArea;

    // instance variable accessible to all methods of this class
    int x = 1;

    // set up applet's GUI
    public void init()
    {
        outputArea = new JTextArea();
        Container container = getContentPane();
        container.add(outputArea);
    } // end method init

    // method start called after init completes; start calls
    // methods useLocal and useInstance
    public void start()
    {
        int x = 5; // variable local to method start

        outputArea.append("local x in start is \n \n \n \n x");

        useLocal(); // useLocal has local x
        useInstance(); // useInstance uses instance variable x
        useLocal(); // useLocal reinitializes local x
        useInstance(); // instance variable x retains its value

        outputArea.append("\n\nlocal x in start is \n \n \n \n + x");
    } // end method start

    // useLocal reinitializes local variable x during each call
    public void useLocal()
    {
        int x = 25; // initialized each time useLocal is called

        outputArea.append("\n\nlocal x in useLocal is \n \n \n \n + x + \n \n \n \n + x");
        ++x;
        outputArea.append("\n\nlocal x in useLocal is \n \n \n \n + x + \n \n \n \n before exiting useLocal");
    } // end method useLocal
Chapter 6 Methods 277

Line 14 declares and initializes instance variable x to 1. This instance variable is hidden in any block (or method) that declares a variable named x. Method **start** declares a local variable x (line 29) and initializes it to 5. This variable’s value is displayed in the JTextArea outputArea to show that the instance variable x is hidden in **start**. The program defines two other methods—**useLocal** (lines 43–53) and **useInstance** (lines 56–64)—that each take no arguments and do not return results. Method **start** calls each of these methods twice. Method **useLocal** defines local (automatic) variable x (line 45). When **useLocal** is called (line 33), it creates local variable x and initializes x to 25, displays the value of x in outputArea, increments x and displays the value of x again. When **useLocal** is called again (line 35), it recreates local variable x and initializes x to 25. Method **useInstance** does not declare any variables. Therefore, when it refers to variable x, the instance variable x is used. When **useInstance** is called (line 34), it displays the instance variable x in outputArea, multiplies the instance variable x by 10 and displays instance variable x again before returning. The next time method **useInstance** is called (line 36), the instance variable has its modified value, 10. Finally, the program displays the local variable x in **start** again to show that none of the method calls modified the value of x, because the methods all referred to variables in other scopes.
6.12 Recursion

The programs we have discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it is useful to have methods call themselves. A recursive method is a method that calls itself either directly or indirectly through another method. Recursion is an important topic discussed at length in upper-level computer science courses. In this and the next section, simple examples of recursion are presented. This book contains an extensive treatment of recursion. Figure 6.15 (at the end of Section 6.14) summarizes the recursion examples and exercises in this book.

We consider recursion conceptually first. Then we examine several programs containing recursive methods. Recursive problem-solving approaches have a number of elements in common. A recursive method is called to solve a problem. The method actually knows how to solve only the simplest case(s) or so-called base case(s). If the method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method divides the problem into two conceptual pieces: a piece that the method knows how to do (base case) and a piece that the method does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of the original problem. Because this new problem looks like the original problem, the method invokes (calls) a fresh copy of itself to go to work on the smaller problem; this procedure is referred to as a recursive call and is also called the recursion step. The recursion step also normally includes the keyword return, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still open (i.e., while it has not finished executing). The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces. For the recursion eventually to terminate, each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case. At that point, the method recognizes the base case, and returns a result to the previous copy of the method, and a sequence of returns ensues up the line until the original method call eventually returns the final result to the caller. This process sounds exotic when compared with the conventional problem solving we have performed to this point. As an example of these concepts at work, let us write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer \( n \), written \( n! \) (and pronounced “\( n \) factorial”), is the product

\[
 n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1
\]

where \( 1! \) is equal to 1 and \( 0! \) is defined to be 1. For example, \( 5! \) is the product \( 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \), which is equal to 120.

The factorial of an integer, \texttt{number}, greater than or equal to 0, can be calculated iteratively (nonrecursively) using the \texttt{for} structure as follows:

```java
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
    factorial *= counter;
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
A recursive definition of the factorial method is arrived at by observing the following relationship:

\[ n! = n \cdot (n - 1)! \]

For example, 5! is clearly equal to 5 * 4!, as is shown by the following equations:

\[ 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \]
\[ 5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \]
\[ 5! = 5 \cdot (4!) \]

The evaluation of 5! would proceed as shown in Fig. 6.11. Figure 6.11 (a) shows how the succession of recursive calls proceeds until 1! is evaluated to 1, which terminates the recursion. Figure 6.11 (b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 6.12 uses recursion to calculate and print the factorials of the integers from 0 to 10. (The choice of the data type `long` will be explained momentarily.) The recursive method `factorial` (lines 29–39) first tests to determine whether a terminating condition (line 32) is true. If `number` is less than or equal to 1 (the base case), `factorial` returns 1, no further recursion is necessary and the method returns. If `number` is greater than 1, line 37 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`. Note that `factorial(number - 1)` is a slightly simpler problem than the original calculation `factorial(number)`.

---

```java
1 // Fig. 6.12: FactorialTest.java
2 // Recursive factorial method
3
4 // Java core packages
5 import java.awt.*;
```

---

Fig. 6.11  Recursive evaluation of 5!.

```
(a) Procession of recursive calls.  (b) Values returned from each recursive call.
```

```
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5! = 5 * 24 = 120 is returned
4! = 4 * 6 = 24 is returned
3! = 3 * 2 = 6 is returned
2! = 2 * 1 = 2 is returned
1 is returned
```

Fig. 6.12  Calculating factorials with a recursive method (part 1 of 2).
© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
Method `factorial` (line 29) receives a parameter of type `long` and returns a result of type `long`. As can be seen in Fig. 6.12, factorial values become large quickly. We chose data type `long` so the program can calculate factorials greater than 20!. Unfortunately, the
factorial method produces large values so quickly that even long does not help us print many factorial values before the values exceed the size that can be stored in a long variable.

We explore in the exercises the fact that float and double may ultimately be needed by users desiring to calculate factorials of larger numbers. This situation points to a weakness in most programming languages, namely, that the languages are not easily extended to handle the unique requirements of various applications. As we will see in Chapter 9, Object-Oriented Programming, Java is an extensible language that allows us to create arbitrarily large integers if we wish. In fact, package java.math provides two classes—BigInteger and BigDecimal—explicitly for mathematical calculations of arbitrary precision that cannot be represented with Java’s primitive data types.

6.13 Example Using Recursion: The 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 series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…. This number, too, repeatedly occurs in nature and has been called the golden ratio, or the golden mean. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden-mean length/width ratio.

The Fibonacci series may be defined recursively as follows:

\[
\begin{align*}
\text{fibonacci}(0) &= 0 \\
\text{fibonacci}(1) &= 1 \\
\text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)
\end{align*}
\]

Note that there are two base cases for the Fibonacci calculation: fibonacci(0) is defined to be 0, and fibonacci(1) is defined to be 1. The applet of Fig. 6.13 calculates the \(i\)th Fibonacci number recursively, using method fibonacci (lines 68–78). The applet enables the user to input an integer in a JTextField. The value input indicates the \(i\)th Fibonacci number to calculate. When the user presses the Enter key, method actionPerformed executes in response to the user interface event and calls method fibonacci to calculate the specified Fibonacci number. Fibonacci numbers tend to become large quickly. Therefore, we use data type long as the parameter type and the return type of fibonacci. In Fig. 6.13, the screen captures show the results of calculating several Fibonacci numbers.

Once again, method init of this applet creates the GUI components and attaches them to the applet’s content pane. The layout manager for the content pane is set to FlowLayout at line 22.
The event handling in this example is similar to the event handling of the Craps applet in Fig. 6.9. Line 34 specifies that this applet should listen for events from the JTextField numberField. Remember, the this keyword enables the applet to refer to itself. So, in line 34, the applet is telling numberField that the applet should be notified (with a call to the applet’s actionPerformed method) when an action event occurs in the numberField. In this example, the user presses the Enter key while typing in the numberField to generate the action event. A message is then sent to the applet (i.e., a method—actionPerformed—is called on the applet) indicating that the user of the program has interacted with one of the program’s GUI components (numberField). Remember that the statement to register the applet as the numberField’s listener will compile only if the applet class also implements ActionListener (line 12).

```java
// Fig. 6.13: FibonacciTest.java
// Recursive fibonacci method

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

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

public class FibonacciTest extends JApplet
    implements ActionListener {
    JLabel numberLabel, resultLabel;
    JTextField numberField, resultField;

    // set up applet’s GUI
    public void init() {
        // obtain content pane and set its layout to FlowLayout
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // create numberLabel and attach it to content pane
        numberLabel =
            new JLabel( "Enter an integer and press Enter" );
        container.add( numberLabel );

        // create numberField and attach it to content pane
        numberField = new JTextField( 10 );
        container.add( numberField );

        // register this applet as numberField’s ActionListener
        numberField.addActionListener( this );

        // create resultLabel and attach it to content pane
        resultLabel = new JLabel( "Fibonacci value is" );
        container.add( resultLabel );
    }

    // method that does the actual fibonacci calculation
    public int fibonacci( int n ) {
        if ( n == 0 ) return 0;
        if ( n == 1 ) return 1;
        return fibonacci( n-1 ) + fibonacci( n-2 );
    }

    // method that displays fibonacci value
    public void actionPerformed( ActionEvent event ) {
        int n = Integer.parseInt( numberField.getText() );
        int result = fibonacci( n );
        resultLabel.setText( "Fibonacci value is " + Integer.toString( result ) );
    }
}
```

Fig. 6.13 Recursively generating Fibonacci numbers (part 1 of 3).
```java
// create numberField, make it uneditable
// and attach it to content pane
resultField = new JTextField( 15 );
resultField.setEditable( false );
container.add( resultField );

} // end method init

// obtain user input and call method fibonacci
public void actionPerformed( ActionEvent e )
{
    long number, fibonacciValue;

    // obtain user's input and convert to long
    number = Long.parseLong( numberField.getText() );

    showStatus( "Calculating ..." );

    // calculate fibonacci value for number user input
    fibonacciValue = fibonacci( number );

    // indicate processing complete and display result
    showStatus( "Done." );
    resultField.setText( Long.toString( fibonacciValue ) );

} // end method actionPerformed

// Recursive definition of method fibonacci
public long fibonacci( long n )
{
    // base case
    if ( n == 0 || n == 1 )
        return n;

    // recursive step
    else
        return fibonacci( n - 1 ) + fibonacci( n - 2 );

} // end method fibonacci
```

Fig. 6.13 Recursively generating Fibonacci numbers (part 2 of 3).
The call to `fibonacci` (line 59) from `actionPerformed` is not a recursive call, but all subsequent calls to `fibonacci` performed from the body of `fibonacci` are recursive. Each time `fibonacci` is invoked, it immediately tests for the base case—`n` equal to 0 or 1. If this condition is true, `fibonacci` returns `n` (fibonacci(0) is 0, and fibonacci(1) is 1). Interestingly, if `n` is greater than 1, the recursion step generates two recursive calls, each for a slightly simpler problem than the original call to `fibonacci`.

Fig. 6.13  Recursively generating Fibonacci numbers (part 3 of 3).
Figure 6.14 shows how method \texttt{fibonacci} evaluates \texttt{fibonacci(3)}. In the figure, \( f \) is an abbreviation for \texttt{fibonacci}.

Figure 6.14 raises some interesting issues about the order in which Java compilers evaluate the operands of operators. This issue is different than the order in which operators are applied to their operands, namely the order dictated by the rules of operator precedence. From Fig. 6.14, it appears that while \( f(3) \) is being evaluated, two recursive calls will be made, namely \( f(2) \) and \( f(1) \). But in what order will these calls be made? Most programmers assume that the operands will be evaluated left to right. In Java, this assumption is true.

The C and C++ languages (on which many of Java’s features are based) do not specify the order in which the operands of most operators (including \(+\)) are evaluated. Therefore, the programmer can make no assumption in those languages about the order in which the calls in this example execute. The calls could, in fact, execute \( f(2) \) first and \( f(1) \) second, or the calls could be executed in the reverse order: \( f(1) \), then \( f(2) \). In this program and in most other programs, it turns out that the final result would be the same for either case. But in some programs, the evaluation of an operand may have \textit{side effects} that could affect the final result of the expression.

The Java language specifies that the order of evaluation of the operands is from left to right. Thus, the method calls are in fact \( f(2) \) first and \( f(1) \) second.

\textbf{Good Programming Practice 6.8}

Do not write expressions that depend on the order of evaluation of the operands of an operator. Use of such expressions often results in programs that are difficult to read, debug, modify and maintain.

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each invocation of the \texttt{fibonacci} method that does not match one of the base cases (i.e., 0 or 1) results in two more recursive calls to the \texttt{fibonacci} method. This set of recursive calls rapidly gets out of hand. Calculating the Fibonacci value of 20 using the program in Fig. 6.13 requires 21,891 calls to the \texttt{fibonacci} method; calculating the Fibonacci value of 30 requires 2,692,537 calls to the \texttt{fibonacci} method.

![Fig. 6.14 Set of recursive calls to method \texttt{fibonacci} (f in this diagram).](image-url)
As you try calculate larger Fibonacci values, you will notice that each consecutive Fibonacci number you ask the applet to calculate results in a substantial increase in calculation time and number of calls to the \texttt{fibonacci} method. For example, the Fibonacci value of 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls. As you can see, the number of calls to \texttt{fibonacci} is increasing quickly—1,664,080 additional calls between Fibonacci values of 30 and 31 and 2,692,538 additional calls between Fibonacci values of 31 and 32. This difference in number of calls made between Fibonacci values of 31 and 32 is more than 1.5 times the number of calls for Fibonacci values between 30 and 31. Problems of this nature humble even the world’s most powerful computers! In the field of complexity theory, computer scientists study how hard algorithms work to complete their tasks. Complexity issues are discussed in detail in the upper-level computer science curriculum course generally called “Algorithms.”

\textbf{Performance Tip 6.3}

Avoid Fibonacci-style recursive programs, which result in an exponential “explosion” of calls.

\textbf{Testing and Debugging Tip 6.3}

Try enhancing the Fibonacci program of Fig. 6.13 such that it calculates the approximate amount of time required to perform the calculation. For this purpose, call \texttt{static System}\ method \texttt{getCurrentTimeMillis}, which takes no arguments and returns the computer’s current time in milliseconds. Call this method twice—once before the call to \texttt{fibonacci} and once after the call to \texttt{fibonacci}. Save each of these values and calculate the difference in the times to determine how many milliseconds were required to perform the calculation. Display this result.

\section*{6.14 Recursion vs. Iteration}

In the previous sections, we studied two methods that can easily be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why the programmer might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure (such as \texttt{for}, \texttt{while} or \texttt{do/while}); recursion uses a selection structure (such as \texttt{if}, \texttt{if/else} or \texttt{switch}). Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Recursion has many negatives. It repeatedly invokes the mechanism and, consequently the overhead, of method calls. This repetition can be expensive in terms of both processor time and memory space. Each recursive call causes another copy of the method (actually, only the method’s variables) to be created; this set of copies can consume considerable
memory space. Iteration normally occurs within a method, so the overhead of repeated method calls and extra memory assignment is omitted. So why choose recursion?

Software Engineering Observation 6.10
Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Often, a recursive approach can be implemented with few lines of code and a corresponding iterative approach may take large amounts of code. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

Performance Tip 6.4
Avoid using recursion in situations requiring performance. Recursive calls take time and consume additional memory.

Common Programming Error 6.16
Accidentally having a nonrecursive method call itself either directly or indirectly through another method can cause infinite recursion.

Most programming textbooks introduce recursion much later than we have done here. We feel that recursion is a sufficiently rich and complex topic that it is better to introduce it earlier and spread examples of it over the remainder of the text. Figure 6.15 summarizes the recursion examples and exercises in this text.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Recursion examples and exercises</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Factorial method</td>
</tr>
<tr>
<td></td>
<td>Fibonacci method</td>
</tr>
<tr>
<td></td>
<td>Greatest common divisor</td>
</tr>
<tr>
<td></td>
<td>Sum of two integers</td>
</tr>
<tr>
<td></td>
<td>Multiply two integers</td>
</tr>
<tr>
<td></td>
<td>Raising an integer to an integer power</td>
</tr>
<tr>
<td></td>
<td>Towers of Hanoi</td>
</tr>
<tr>
<td></td>
<td>Visualizing recursion</td>
</tr>
<tr>
<td>7</td>
<td>Sum the elements of an array</td>
</tr>
<tr>
<td></td>
<td>Print an array</td>
</tr>
<tr>
<td></td>
<td>Print an array backward</td>
</tr>
<tr>
<td></td>
<td>Check if a string is a palindrome</td>
</tr>
<tr>
<td></td>
<td>Minimum value in an array</td>
</tr>
<tr>
<td></td>
<td>Selection sort</td>
</tr>
<tr>
<td></td>
<td>Eight Queens</td>
</tr>
<tr>
<td></td>
<td>Linear search</td>
</tr>
<tr>
<td></td>
<td>Binary search</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
</tr>
<tr>
<td></td>
<td>Maze traversal</td>
</tr>
<tr>
<td>10</td>
<td>Printing a string input at the keyboard backward</td>
</tr>
</tbody>
</table>

Fig. 6.15  Summary of recursion examples and exercises in this text (part 1 of 2).
Let us reconsider some observations we make repeatedly throughout this book. Good software engineering is important. High performance is often important. Unfortunately, these goals are often at odds with one another. Good software engineering is key to making more manageable the task of developing larger and more complex software systems. High performance in these systems is key to realizing the systems of the future, which will place ever greater computing demands on hardware. Where do methods fit in here?

**Software Engineering Observation 6.11**
Modularizing programs in a neat, hierarchical manner promotes good software engineering. But it has a price.

**Performance Tip 6.5**
A heavily modularized program—as compared with a monolithic (i.e., one-piece) program without methods—makes potentially large numbers of method calls, which consume execution time and space on a computer's processor(s). But monolithic programs are difficult to program, test, debug, maintain and evolve.

So modularize your programs judiciously, always keeping in mind the delicate balance between performance and good software engineering.

### 6.15 Method Overloading

Java enables several methods of the same name to be defined, as long as the methods have different sets of parameters (based on the number of parameters, the types of the parameters and the order of the parameters). This characteristic is called method overloading. When an overloaded method is called, the Java compiler selects the proper method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the same name that perform similar tasks, but on different data types.

**Good Programming Practice 6.9**
Overloading methods that perform closely related tasks can make programs more readable and understandable.

Figure 6.16 uses overloaded method `square` to calculate the square of an `int` and the square of a `double`. 
```java
// Fig. 6.16: MethodOverload.java
// Using overloaded methods

// Java core packages
import java.awt.Container;

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

public class MethodOverload extends JApplet {

    // set up GUI and call versions of method square
    public void init() {
        JTextArea outputArea = new JTextArea();
        Container container = getContentPane();
        container.add(outputArea);

        outputArea.setText("The square of integer 7 is " + square(7) + 
                         "The square of double 7.5 is " + square(7.5));
    }

    // square method with int argument
    public int square(int intValue) {
        System.out.println("Called square with int argument: " + intValue);
        return intValue * intValue;
    }

    // square method with double argument
    public double square(double doubleValue) {
        System.out.println("Called square with double argument: " + doubleValue);
        return doubleValue * doubleValue;
    }

    // end class MethodOverload
}
```

Fig. 6.16 Using overloaded methods (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
Overloaded methods are distinguished by their signature—a combination of the method’s name and its parameter types. If the Java compiler looked only at method names during compilation, the code in Fig. 6.16 would be ambiguous—the compiler would not know how to distinguish between the two `square` methods. Logically, the compiler uses longer “mangled” or “decorated” names that include the original method name, the types of each parameter and the exact order of the parameters to determine if the methods in a class are unique in that class.

For example, in Fig. 6.16, the compiler might use the logical name “square of int” for the `square` method that specifies an `int` parameter and “square of double” for the `square` method that specifies a `double` parameter. If a method `foo`’s definition begins as

```java
void foo( int a, float b )
```

then the compiler might use the logical name “foo of int and float.” If the parameters are specified as

```java
void foo( float a, int b )
```

then the compiler might use the logical name “foo of float and int.” Note that the order of the parameters is important to the compiler. The preceding two `foo` methods are considered to be distinct by the compiler.

The logical names of methods used by the compiler did not mention the return types of the methods, because methods cannot be distinguished by return type. The program in Fig. 6.17 illustrates the compiler errors generated when two methods have the same signature and different return types. Overloaded methods can have different return types, but must have different parameter lists. Also, overloaded methods need not have the same number of parameters.

Common Programming Error 6.17

Creating overloaded methods with identical parameter lists and different return types is a syntax error.
6.16 Methods of Class JApplet

We have written many applets to this point in the text, but we have not yet discussed the key methods of class JApplet that the applet container calls during the execution of an applet. Figure 6.18 lists the key methods of class JApplet, specifies when they get called and explains the purpose of each method.

These JApplet methods are defined by the Java API to do nothing unless you provide a definition in your applet’s class definition. If you would like to use one of these methods in an applet you are defining, you must define the first line of each method as shown in Fig. 6.18. Otherwise, the applet container will not call your versions of the methods during the applet’s execution. Defining the methods as discussed here is known as overriding the original method definition. The applet container will call the overridden version of a method for your applet before it attempts to call the default versions inherited from JApplet. Overriding is discussed in detail in Chapter 9.

Common Programming Error 6.18

Providing a definition for one of the JApplet methods init, start, paint, stop or destroy that does not match the method headers shown in Figure 6.18 results in a method that will not be called automatically during execution of the applet.
repaint is also of interest to many applet programmers. The applet’s paint method normally is called by the applet container. What if you would like to change the appearance of the applet in response to the user’s interactions with the applet? In such situations, you may want to call paint directly. However, to call paint, we must pass it the Graphics parameter it expects. This requirement poses a problem for us. We do not have a Graphics object at our disposal to pass to paint. (We discuss this issue in Chapter 18, Multimedia.) For this reason, class JApplet provides method repaint. The statement

```java
repaint();
```
obtains the `Graphics` object for you and invokes another method, called `update`. Method `update` invokes method `paint` and passes to it the `Graphics` object. The `repaint` method is discussed in detail in Chapter 18, “Multimedia.”

### 6.17 (Optional Case Study) Thinking About Objects: Identifying Class Operations

In the “Thinking About Objects” sections at the ends of Chapters 3, 4 and 5, we performed the first few steps in the object-oriented design for our elevator simulator. In Chapter 3, we identified the classes we need to implement. In Chapter 4, we created a class diagram that models the structure of our system. In Chapter 5, we examined objects’ states and modeled objects’ activities and state transitions.

In this section, we concentrate on determining the class operations (or behaviors) needed to implement the elevator simulator. In Chapter 7, we concentrate on the collaborations (interactions) between objects of our classes.

An operation of a class is a service that the class provides to “clients” (users) of that class. Consider the operations of some real-world classes. A radio’s operations include setting its station and volume (typically invoked by a person adjusting the radio’s controls). A car’s operations include accelerating (invoked by the driver pressing the accelerator pedal), decelerating (invoked by the driver pressing the brake pedal and/or releasing the gas pedal), turning and shifting gears.

We can derive many of the operations of each class directly from the problem statement. To do so, we examine the verbs and verb phrases in the problem statement. We then relate each of these to particular classes in our system (Fig. 6.20). Many of the verb phrases in Fig. 6.20 help us determine the operations of our classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Verb phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>moves to other floor, arrives at a floor, resets elevator button, rings elevator bell, signals its arrival, opens its door, closes its door</td>
</tr>
<tr>
<td>ElevatorShaft</td>
<td>turns off light, turns on light, resets floor button</td>
</tr>
<tr>
<td>Person</td>
<td>walks on floor, presses floor button, presses elevator button, rides elevator, enters elevator, exits elevator</td>
</tr>
<tr>
<td>Floor</td>
<td>[none in the problem statement]</td>
</tr>
<tr>
<td>FloorButton</td>
<td>requests elevator</td>
</tr>
<tr>
<td>ElevatorButton</td>
<td>closes elevator door, signals elevator to move to opposite floor</td>
</tr>
<tr>
<td>FloorDoor</td>
<td>signals person to enter elevator (by opening)</td>
</tr>
<tr>
<td>ElevatorDoor</td>
<td>signals person to exit elevator (by opening), opens floor door, closes floor door</td>
</tr>
<tr>
<td>Bell</td>
<td>[none in the problem statement]</td>
</tr>
<tr>
<td>Light</td>
<td>[none in the problem statement]</td>
</tr>
<tr>
<td>ElevatorModel</td>
<td>creates person</td>
</tr>
</tbody>
</table>

**Fig. 6.20** Verb phrases for each class in simulator.
To create operations, we examine the verb phrases listed with each class. The phrase “moves to other floor” listed with class **Elevator** refers to the activity in which the elevator moves between floors. Should “moves” be an operation of class **Elevator**? The elevator decides to move in response to a button press. A button signals the elevator to move, but a button does not actually move the elevator—therefore, “moves to other floor” does not correspond to an operation. (We include the operations for informing the elevator to move to the other floor later in the discussion, when we discuss the verb phrases associated with the buttons.) The “arrives at a floor” phrase is also not an operation, because the elevator itself decides when to arrive on the floor after five seconds of travel.

The “resets elevator button” phrase associated with class **Elevator** implies that the elevator informs the elevator button to reset. Therefore, class **ElevatorButton** needs an operation to provide this service to the elevator. We place this operation (**resetButton**) in the bottom compartment of class **ElevatorButton** in our class diagram (Fig. 6.21).

We represent the names of the operations as method names (by following the names with a pair of parentheses) and include the return type after the colon:

```
resetButton() : void
```

The parentheses can contain a comma-separated list of the parameters that the operation takes—in this case, none. For the moment, most of our operations have no parameters and a **void** return type; this might change as our design and implementation processes proceed.

---

**Fig. 6.21** Classes with attributes and operations.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
From the “ring the elevator bell” phrase listed with class Elevator, we conclude that class Bell should have an operation that provides a service—namely, ringing. We list the ringBell operation under class Bell.

When arriving at a floor, the elevator “signals its arrival” to the doors. The elevator door responds by opening, as implied by the phrase “opens [the elevator’s] door” associated with class Elevator. Therefore, class ElevatorDoor needs an operation that opens its door. We place the openDoor operation in the bottom compartment of this class. The phrase “closes [the elevator’s] door” indicates that class ElevatorDoor needs an operation that closes its door, so we place the closeDoor operation in the same compartment.

Class ElevatorShaft lists “turns off light” and “turns on light” in its verb-phrases column, so we create the turnOffLight and turnOnLight operations and list them under class Light. The “resets floor button” phrase implies that the elevator instructs a floor button to reset. Therefore, class FloorButton needs a resetButton operation.

The phrase “walks on floor” listed by class Person is not an operation, because a person decides to walk across the floor in response to that person’s creation. However, the phrases “presses floor button” and “presses elevator button” are operations pertaining to the button classes. We therefore place the pressButton operation under classes FloorButton and ElevatorButton in our class diagram (Fig. 6.21). The phrase “rides elevator” implies that Elevator needs a method that allows a person to ride the elevator, so we place operation ride in the bottom compartment of Elevator. The “enters elevator” and “exits elevator” phrases listed with class Person suggest that class Elevator needs operations that correspond to these actions. We place operations enterElevator and exitElevator in the bottom compartment of class Elevator.

The “requests elevator” phrase listed under class FloorButton implies that class Elevator needs a requestElevator operation. The phrase “signals elevator to move to opposite floor” listed with class ElevatorButton implies that ElevatorButton informs Elevator to depart. Therefore, the Elevator needs to provide a “departure” service; we place a departElevator operation in the bottom compartment of Elevator.

The phrases listed with classes FloorDoor and ElevatorDoor mention that the doors—by opening—signal a Person object to enter or exit the elevator. Specifically, a door informs a person that the door has opened. (The person then enters or exits the elevator, accordingly.) We place the doorOpened operation in the bottom compartment for class Person. In addition, the ElevatorDoor opens and closes the FloorDoor, so we assign openDoor and closeDoor to the bottom compartment of class FloorDoor.

Lastly, the “creates person” action associated with class ElevatorModel refers to creating a Person object and adding it to the simulation. Although we can require ElevatorModel to send a “create person” and an “add person” message, an object of class Person cannot respond to these messages, because that object does not yet exist. We discuss new objects when we consider implementation in Chapter 8. We place the operation addPerson in the bottom compartment of ElevatorModel in the class diagram of Fig. 6.21 and anticipate that the application user will invoke this operation.

---

1. At this point, we can only guess what these operations do. For example, perhaps these operations model real-world elevators, some of which have sensors that detect when passengers enter and exit. For now, we simply list these operations. We will discover what, if any, actions these operations perform as we continue our design process.
For now, we do not concern ourselves with operation parameters or return types; we attempt to gain only a basic understanding of the operations of each class. As we continue our design process, the number of operations belonging to each class may vary—we might find that new operations are needed or that some current operations are unnecessary—and we might determine that some of our class operations need non-void return types.

**SUMMARY**

- The best way to develop and maintain a large program is to divide it into several smaller modules. Modules are written in Java as classes and methods.
- A method is invoked by a method call. The method call mentions the method by name and provides arguments in parentheses that the called method requires to perform its task. If the method is in another class, the call must be preceded by a reference name and a dot operator. If the method is static, it must be preceded by a class name and a dot operator.
- Each argument of a method may be a constant, a variable or an expression.
- A local variable is known only in a method definition. Methods are not allowed to know the implementation details of any other method (including its local variables).
- The on-screen display area for a JApplet has a content pane to which the GUI components must be attached so they can be displayed at execution time. The content pane is an object of class Container from the java.awt package.
- Method `getContentPane` of class JApplet returns a reference to the applet’s content pane.
- The general format for a method definition is

  ```java
  return-value-type method-name ( parameter-list )
  {
    declarations and statements
  }
  ```

  The return-value-type states the type of the value returned to the calling method. If a method does not return a value, the return-value-type is void. The method-name is any valid identifier. The parameter-list is a comma-separated list containing the declarations of the variables that will be passed to the method. If a method does not receive any values, parameter-list is empty. The method body is the set of declarations and statements that constitute the method.
- The arguments passed to a method should match in number, type and order with the parameters in the method definition.
- When a program encounters a method, control transfers from the point of invocation to the called method, the method executes and control returns to the caller.
- A called method can return control to the caller in one of three ways. If the method does not return a value, control returns at the method-ending right brace or by executing the statement

  ```java
  return;
  ```

  If the method does return a value, the statement

  ```java
  return expression;
  ```

  returns the value of expression.
- There are three ways to call a method—the method name by itself, a reference to an object followed by the dot (.) operator and the method name, and a class name followed by the dot (.) operator and a method name. The last syntax is for static methods of a class.
• An important feature of method definitions is the coercion of arguments. In many cases, argument values that do not correspond precisely to the parameter types in the method definition are converted to the proper type before the method is called. In some cases, these conversions can lead to compiler errors if Java’s promotion rules are not followed.

• The promotion rules specify how types can be converted to other types without losing data. The promotion rules apply to mixed-type expressions. The type of each value in a mixed-type expression is promoted to the “highest” type in the expression.

• Method `Math.random` generates a double value from 0.0 up to, but not including, 1.0. Values produced by `Math.random` can be scaled and shifted to produce values in a range.

• The general equation for scaling and shifting a random number is

\[
n = a + \text{(int)} \left( \text{Math.random()} \times b \right);
\]

where \(a\) is the shifting value (the first number in the desired range of consecutive integers) and \(b\) is the scaling factor (the width of the desired range of consecutive integers).

• A class can inherit existing attributes and behaviors (data and methods) from another class specified to the right of keyword `extends` in the class definition. In addition, a class can implement one or more `interfaces`. An interface specifies one or more behaviors (i.e., methods) that you must define in your class definition.

• The interface `ActionListener` specifies that a class must define a method with the first line

```java
public void actionPerformed( ActionEvent actionEvent )
```

• The task of method `actionPerformed` is to process a user’s interaction with a GUI component that generates an action event. This method is called in response to the user interaction (the event). This process is called `event handling`. The event handler is the `actionPerformed` method, which is called in response to the event. This style of programming is known as `event-driven programming`.

• Keyword `final` declares constant variables. Constant variables must be initialized before they are used in a program. Constant variables are often called `named constants` or `read-only variables`.

• A `JLabel` contains a string of characters to be displayed on the screen. Normally, a `JLabel` indicates the purpose of another GUI element on the screen.

• `JTextFields` get information from the user or displays information on the screen.

• When the user presses a `JButton`, the program normally responds by performing a task.

• `Container` method `setLayout` defines the layout manager for the applet’s user interface. Layout managers are provided to arrange GUI components on a `Container` for presentation purposes.

• `FlowLayout` is the simplest layout manager. GUI components are placed on a `Container` from left to right in the order in which they are attached to the `Container` with method `add`. When the edge of the container is reached, components are continued on the next line.

• Before any event can be processed, each GUI component must know which object in the program defines the event-handling method that will be called when an event occurs. Method `addActionListener` is used to tell a `JButton` or `JTextField` that another object is listening for action events and defines method `actionPerformed`. This procedure is called `registering the event handler with the GUI component`. To respond to an action event, we must define a class that implements `ActionListener` and defines method `actionPerformed`. Also, we must register the event handler with the GUI component.

• Method `showStatus` displays a `String` in the applet container’s status bar.
• Each variable identifier has the attributes duration (lifetime) and scope. An identifier’s duration determines when that identifier exists in memory. An identifier’s scope is where the identifier can be referenced in a program.

• Identifiers that represent local variables in a method have automatic duration. Automatic-duration variables are created when program control reaches their declaration; they exist while the block in which they are declared is active; and they are destroyed when the block in which they are declared is exited.

• Java also has identifiers of static duration. Variables and references of static duration exist from the point at which the class in which they are defined is loaded into memory for execution until the program terminates.

• The scopes for an identifier are class scope and block scope. An instance variable declared outside any method has class scope. Such an identifier is “known” in all methods of the class. Identifiers declared inside a block have block scope. Block scope ends at the terminating right brace (}) of the block.

• Local variables declared at the beginning of a method have block scope, as do method parameters, which are considered to be local variables of the method.

• Any block may contain variable declarations.

• A recursive method is a method that calls itself, either directly or indirectly.

• If a recursive method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method divides the problem into two or more conceptual pieces: A piece that the method knows how to do and a slightly smaller version of the original problem. Because this new problem looks like the original problem, the method launches a recursive call to work on the smaller problem.

• For recursion to terminate, the sequence of smaller and smaller problems must converge to the base case. When the method recognizes the base case, the result is returned to the previous method call, and a sequence of returns ensues all the way up the line until the original call of the method returns the final result.

• Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.

• Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls.

• Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.

• Iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges to the base case.

• Recursion repeatedly invokes the mechanism and, consequently the overhead, of method calls. This repetition can be expensive in terms of both processor time and memory space.

• The user presses the Enter key while typing in a JTextField to generate an action event. The event handling for this GUI component is set up like a JButton: A class must be defined that implements ActionListener and defines method actionPerformed. Also, the JTextField’s addActionListener method must be called to register the event.

• It is possible to define methods with the same name, but different parameter lists. This feature is called method overloading. When an overloaded method is called, the compiler selects the proper method by examining the arguments in the call.

• Overloaded methods can have different return values and must have different parameter lists. Two methods differing only by return type will result in a syntax error.
• The applet’s init method is called once by the applet container when an applet is loaded for execution. It performs initialization of an applet. The applet’s start method is called after the init method completes execution and every time the user of the browser returns to the HTML page on which the applet resides (after browsing another HTML page).

• The applet’s paint method is called after the init method completes execution and the start method has started executing to draw on the applet. It is also called every time the applet needs to be repainted.

• The applet’s stop method is called when the applet should suspend execution—normally, when the user of the browser leaves the HTML page on which the applet resides.

• The applet’s destroy method is called when the applet is being removed from memory—normally, when the user of the browser exits the browsing session.

• Method repaint can be called in an applet to cause a fresh call to paint. Method repaint invokes another method called update and passes it the Graphics object. The update method invokes the paint method and passes it the Graphics object.

**TERMINOLOGY**

**KeyEvent** class
**JButton** class of package javax.swing
**FlowLayout** class
**JLabel** class of package javax.swing
**JTextField** class of package javax.swing

**Math** class methods
Math.E
Math.PI
Math.random method

argument in a method call
automatic duration
automatic variable
base case in recursion
block
block scope
call a method
called method
caller
calling method
class
class scope
coelection of arguments
constant variable
copy of a value
destroy method of JApplet
divide and conquer
duration
element of chance
factorial method
final
FlowLayout class
init method of JApplet
invoke a method
iteration
Java API (Java class library)
JButton class of package javax.swing
JLabel class of package javax.swing
JTextField class of package javax.swing
setLayout method of JApplet
local variable
method call
method-call operator, ()
method declaration
method definition
method overloading
mixed-type expression
modular program
named constant
overloading
paint method of JApplet
parameter in a method definition
programmer-defined method
promotion rules
random-number generation
read-only variable
recursion
recursion step
recursive call
recursive method
reference parameter
reference types
repaint method of JApplet
return
return-value type
scaling
scope
shifting
SELF-REVIEW EXERCISES

6.1 Fill in the blanks in each of the following statements:
   a) Program modules in Java are called _________ and _________.
   b) A method is invoked with a _________.
   c) A variable known only within the method in which it is defined is called a _________.
   d) The _________ statement in a called method can be used to pass the value of an expression back to the calling method.
   e) The keyword _________ indicates that a method does not return a value.
   f) The _________ of an identifier is the portion of the program in which the identifier can be used.
   g) The three ways to return control from a called method to a caller are _________, _________ and _________.
   h) The _________ method is invoked once when an applet begins execution.
   i) The _________ method produces random numbers.
   j) The _________ method is invoked each time the user of a browser revisits the HTML page on which an applet resides.
   k) The _________ method is invoked to draw on an applet.
   l) Variables declared in a block or in a method’s parameter list are of _________ duration.
   m) The _________ method invokes the applet’s _________ method, which in turn invokes the applet’s _________ method.
   n) The _________ method is invoked for an applet each time the user of a browser leaves an HTML page on which the applet resides.
   o) A method that calls itself either directly or indirectly is a _________ method.
   p) A recursive method typically has two components: one that provides a means for the recursion to terminate by testing for a _________ case and one that expresses the problem as a recursive call for a slightly simpler problem than does the original call.
   q) In Java, it is possible to have various methods with the same name that each operate on different types and/or numbers of arguments. This feature is called method _________.
   r) The _________ qualifier is used to declare read-only variables.

6.2 For the following program, state the scope (either class scope or block scope) of each of the following elements:
   a) the variable x.
   b) the variable y.
   c) the method cube.
   d) the method paint.
   e) the variable yPos.

```java
public class CubeTest extends JApplet {
    int x;

    public void paint( Graphics g )
    {
        int yPos = 25;
    }
}
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
6.3 Write an application that tests if the examples of the math-library method calls shown in Fig. 6.2 actually produce the indicated results.

6.4 Give the method header for each of the following methods:
   a) Method hypotenuse, which takes two double-precision, floating-point arguments side1 and side2 and returns a double-precision, floating-point result.
   b) Method smallest, which takes three integers x, y and z and returns an integer.
   c) Method instructions, which does not take any arguments and does not return a value. [Note: Such methods are commonly used to display instructions to a user.]
   d) Method intToFloat, which takes an integer argument number and returns a floating-point result.

6.5 Find the error in each of the following program segments. Explain how to correct the error.
   a) int g() {
       System.out.println( "Inside method g" );
       h() {
            System.out.println( "Inside method h" );
        }
    }
   b) int sum( int x, int y ) {
       int result;
       result = x + y;
    }
   c) int sum( int n ) {
       if ( n == 0 )
           return 0;
       else
           n + sum( n - 1 );
    }
   d) void f( float a ); {
       float a;
       System.out.println( a );
    }
   e) void product() {
       int a = 6, b = 5, c = 4, result;
       result = a * b * c;
       System.out.println( "Result is " + result );
       return result;
   }
6.6 Write a complete Java applet to prompt the user for the double radius of a sphere, and call method \texttt{sphereVolume} to calculate and display the volume of that sphere using the assignment

\[
\text{volume} = \left( \frac{4.0}{3.0} \right) \times \text{Math.PI} \times \text{Math.pow( radius, 3 )}
\]

The user should input the radius through a 	exttt{JTextField}.

\textbf{ANSWERS TO SELF-REVIEW EXERCISES}

6.1 \begin{itemize}
\item a) methods and classes. \item b) method call. \item c) local variable. \item d) \texttt{return}. \item e) \texttt{void}. \item f) scope. \item g) \texttt{return}; or \texttt{return expression}; or encountering the closing right brace of a method. \item h) \texttt{init}. \item i) \texttt{Math.random}. \item j) \texttt{start}. \item k) \texttt{paint}. \item l) automatic. \item m) \texttt{repaint}. \item n) \texttt{stop}. \item o) recursive. \item p) base. \item q) overloading. \item r) \texttt{final}.
\end{itemize}

6.2 \begin{itemize}
\item a) Class scope. \item b) Block scope. \item c) Class scope. \item d) Class scope. \item e) Block scope.
\end{itemize}

6.3 The following solution demonstrates the \texttt{Math} class methods in Fig. 6.2:

```java
public class MathTest {
    public static void main( String args[] )
    {
        System.out.println( "Math.abs( 23.7 ) = " + Math.abs( 23.7 ) );
        System.out.println( "Math.abs( 0.0 ) = " + Math.abs( 0.0 ) );
        System.out.println( "Math.abs( -23.7 ) = " + Math.abs( -23.7 ) );
        System.out.println( "Math.ceil( 9.2 ) = " + Math.ceil( 9.2 ) );
        System.out.println( "Math.ceil( -9.8 ) = " + Math.ceil( -9.8 ) );
        System.out.println( "Math.cos( 0.0 ) = " + Math.cos( 0.0 ) );
        System.out.println( "Math.exp( 1.0 ) = " + Math.exp( 1.0 ) );
        System.out.println( "Math.exp( 2.0 ) = " + Math.exp( 2.0 ) );
        System.out.println( "Math.floor( 9.2 ) = " + Math.floor( 9.2 ) );
        System.out.println( "Math.floor( -9.8 ) = " + Math.floor( -9.8 ) );
        System.out.println( "Math.log( 2.718282 ) = " + Math.log( 2.718282 ) );
        System.out.println( "Math.log( 7.389056 ) = " + Math.log( 7.389056 ) );
        System.out.println( "Math.max( 2.3, 12.7 ) = " + Math.max( 2.3, 12.7 ) );
        System.out.println( "Math.max( -2.3, -12.7 ) = " + Math.max( -2.3, -12.7 ) );
        System.out.println( "Math.min( 2.3, 12.7 ) = " + Math.min( 2.3, 12.7 ) );
        System.out.println( "Math.min( -2.3, -12.7 ) = " + Math.min( -2.3, -12.7 ) );
    }
}
```

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
6.4  a)    \textbf{double hypotenuse( double side1, double side2 )}
       b)    \textbf{int smallest( int x, int y, int z )}
       c)    \textbf{void instructions()}
       d)    \textbf{float intToFloat( int number )}

6.5  a) Error: Method \textit{h} is defined in method \textit{g}.
     Correction: Move the definition of \textit{h} outside the definition of \textit{g}.
   b) Error: The method is supposed to return an integer, but does not.
     Correction: Delete variable \textit{result}, and place the statement
        \begin{verbatim}
        return x + y;
        \end{verbatim}
     in the method, or add the following statement at the end of the method body:
        \begin{verbatim}
        return result;
        \end{verbatim}
   c) Error: The result of $n + \text{sum}(n - 1)$ is not returned by this recursive method, resulting
     in a syntax error.
     Correction: Rewrite the statement in the \textbf{else} clause as
        \begin{verbatim}
        return n + \text{sum}(n - 1);
        \end{verbatim}
d) Error: Both the semicolon after the right parenthesis that encloses the parameter list and redefining the parameter \texttt{a} in the method definition are incorrect.
Correction: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration \texttt{float a;}.

e) Error: The method returns a value when it is not supposed to.
Correction: Change the return type to \texttt{int}.

6.6 The following solution calculates the volume of a sphere using the radius entered by the user:

```java
// Exercise 6.6: SphereTest.java

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

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

public class SphereTest extends JApplet
    implements ActionListener {

    JLabel promptLabel;
    JTextField inputField;

    public void init() {
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        promptLabel = new JLabel( "Enter sphere radius: " );
        inputField = new JTextField( 10 );
        inputField.addActionListener( this );
        container.add( promptLabel );
        container.add( inputField );
    }

    public void actionPerformed( ActionEvent actionEvent ) {
        double radius = Double.parseDouble( actionEvent.getActionCommand() );
        showStatus( "Volume is " + sphereVolume( radius ) );
    }

    public double sphereVolume( double radius ) {
        double volume = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );
        return volume;
    }
}
```
EXERCISES

6.7 What is the value of `x` after each of the following statements is performed?
   a) `x = Math.abs( 7.5 );`
   b) `x = Math.floor( 7.5 );`
   c) `x = Math.abs( 0.0 );`
   d) `x = Math.ceil( 0.0 );`
   e) `x = Math.abs( -6.4 );`
   f) `x = Math.ceil( -6.4 );`
   g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

6.8 A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 per hour for each hour or part thereof in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write an applet that calculates and displays the parking charges for each customer who parked a car in this garage yesterday. You should enter in a `JTextField` the hours parked for each customer. The program should display the charge for the current customer and should calculate and display the running total of yesterday’s receipts. The program should use the method `calculateCharges` to determine the charge for each customer.

6.9 An application of method `Math.floor` is rounding a value to the nearest integer. The statement

   \[ y = Math.floor( x + .5 ); \]

will round the number `x` to the nearest integer and assign the result to `y`. Write an applet that reads double values and uses the preceding statement to round each of the numbers to the nearest integer. For each number processed, display both the original number and the rounded number.

6.10 `Math.floor` may be used to round a number to a specific decimal place. The statement

   \[ y = Math.floor( x * 10 + .5 ) / 10; \]

rounds `x` to the tenths position (i.e., the first position to the right of the decimal point). The statement

   \[ y = Math.floor( x * 100 + .5 ) / 100; \]

rounds `x` to the hundredths position (i.e., the second position to the right of the decimal point). Write an applet that defines four methods to round a number `x` in various ways:

   a) `roundToInteger( number )`
   b) `roundToTenths( number )`
   c) `roundToHundredths( number )`
   d) `roundToThousandths( number )`

For each value read, your program should display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.
6.11 Answer each of the following questions:
   a) What does it mean to choose numbers “at random?”
   b) Why is the Math.random method useful for simulating games of chance?
   c) Why is it often necessary to scale and/or shift the values produced by Math.random?
   d) Why is computerized simulation of real-world situations a useful technique?

6.12 Write statements that assign random integers to the variable \( n \) in the following ranges:
   a) \( 1 \leq n \leq 2 \)
   b) \( 1 \leq n \leq 100 \)
   c) \( 0 \leq n \leq 9 \)
   d) \( 1000 \leq n \leq 1112 \)
   e) \( -1 \leq n \leq 1 \)
   f) \( -3 \leq n \leq 11 \)

6.13 For each of the following sets of integers, write a single statement that will print a number at random from the set:
   a) \( 2, 4, 6, 8, 10 \)
   b) \( 3, 5, 7, 9, 11 \)
   c) \( 6, 10, 14, 18, 22 \)

6.14 Write a method \( \text{integerPower}(\ base,\ exponent ) \) that returns the value of \( base^exponent \).

For example, \( \text{integerPower}(3, 4) \) calculates \( 3^4 \) (or \( 3 \times 3 \times 3 \times 3 \)). Assume that \( exponent \) is a positive, nonzero integer and that \( base \) is an integer. Method \( \text{integerPower} \) should use \( \text{for} \) or \( \text{while} \) to control the calculation. Do not use any math library methods. Incorporate this method into an applet that reads integer values from JTextField\s for base and exponent from the user and performs the calculation with the \( \text{integerPower} \) method. [Note: Register for event handling on only the second JTextField. The user should interact with the program by typing numbers in both JTextField\s and pressing Enter only in the second JTextField.]

6.15 Define a method \( \text{hypotenuse} \) that calculates the length of the hypotenuse of a right triangle when the other two sides are given (sample data appear in Fig. 6.22). The method should take two arguments of type double and return the hypotenuse as a double. Incorporate this method into an applet that reads values for \( \text{side1} \) and \( \text{side2} \) from JTextField\s and performs the calculation with the \( \text{hypotenuse} \) method. Determine the length of the hypotenuse for each of the following triangles. [Note: Register for event handling on only the second JTextField. The user should interact with the program by typing numbers in both JTextField\s and pressing Enter only in the second JTextField.]

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Side 1</th>
<th>Side 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>5.0</td>
<td>12.0</td>
</tr>
<tr>
<td>3</td>
<td>8.0</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Fig. 6.22 Values for the sides of triangles in Exercise 6.15.
6.16 Write a method `multiple` that determines for a pair of integers whether the second integer is a multiple of the first. The method should take two integer arguments and return `true` if the second is a multiple of the first and `false` otherwise. Incorporate this method into an applet that inputs a series of pairs of integers (one pair at a time using `JTextField`s). [Note: Register for event handling on only the second `JTextField`. The user should interact with the program by typing numbers in both `JTextField` and pressing `Enter` only in the second `JTextField`.

6.17 Write a method `isEven` that uses the modulus operator to determine if an integer is even. The method should take an integer argument and return `true` if the integer is even and `false` otherwise. Incorporate this method into an applet that inputs a series integers (one at a time using a `JTextField`).

6.18 Write a method `squareOfAsterisks` that displays a solid square of asterisks whose side is specified in integer parameter `side`. For example, if `side` is 4, the method displays

```
****
****
****
****
```

Incorporate this method into an applet that reads an integer value for `side` from the user at the keyboard and performs the drawing with the `squareOfAsterisks` method. Note that this method should be called from the applet’s `paint` method and should be passed the `Graphics` object from `paint`.

6.19 Modify the method created in Exercise 6.18 to form the square out of whatever character is contained in character parameter `fillCharacter`. Thus, if `side` is 5 and `fillCharacter` is “#”, the method should print

```
#####
#####
#####
#####
#####
```

6.20 Use techniques similar to those developed in Exercise 6.18 and Exercise 6.19 to produce a program that graphs a wide range of shapes.

6.21 Modify the program of Exercise 6.18 to draw a solid square with the `fillRect` method of the `Graphics` class. Method `fillRect` receives four arguments: x-coordinate, y-coordinate, width and height. Allow the user to input the coordinates at which the square should appear.

6.22 Write program segments that accomplish each of the following tasks:
   a) Calculate the integer part of the quotient when integer `a` is divided by integer `b`.
   b) Calculate the integer remainder when integer `a` is divided by integer `b`.
   c) Use the program pieces developed in parts a) and b) to write a method `displayDigits` that receives an integer between 1 and 99999 and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer 4562 should be printed as

   `4  5  6  2`

   d) Incorporate the method developed in part c) into an applet that inputs an integer from an input dialog and invokes `displayDigits` by passing the method the integer entered. Display the results in a message dialog.
Implement the following integer methods:

a) Method `celsius` returns the Celsius equivalent of a Fahrenheit temperature, using the calculation

\[
C = \frac{5.0}{9.0} \times (F - 32);
\]

b) Method `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature, using the calculation

\[
F = \frac{9.0}{5.0} \times C + 32;
\]

c) Use these methods to write an applet that enables the user to enter either a Fahrenheit temperature and display the Celsius equivalent or enter a Celsius temperature and display the Fahrenheit equivalent.

[Note: This applet will require two `JTextField` objects that have registered action events. When `actionPerformed` is invoked, the `ActionEvent` parameter has method `getSource()` to determine the GUI component with which the user interacted. Your `actionPerformed` method should contain an `if/else` structure of the form

```java
if (actionEvent.getSource() == input1) {
    // process input1 interaction here
} else {
    // process input2 interaction here
}
```
where `input1` and `input2` are `JTextField` references.]

Write a method `minimum3` that returns the smallest of three floating-point numbers. Use the `Math.min` method to implement `minimum3`. Incorporate the method into an applet that reads three values from the user and determines the smallest value. Display the result in the status bar.

An integer number is said to be a perfect number if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number, because 6 = 1 + 2 + 3. Write a method `perfect` that determines if parameter `number` is a perfect number. Use this method in an applet that determines and displays all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the computing power of your computer by testing numbers much larger than 1000. Display the results in a `JTextArea` that has scrolling functionality.

An integer is said to be prime if it is divisible only by 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

a) Write a method that determines if a number is prime.

b) Use this method in an applet that determines and prints all the prime numbers between 1 and 10,000. How many of these 10,000 numbers do you really have to test before being sure that you have found all the primes? Display the results in a `JTextArea` that has scrolling functionality.

c) Initially, you might think that \( n/2 \) is the upper limit for which you must test to see if a number is prime, but you need only go as high as the square root of \( n \). Why? Rewrite the program, and run it both ways. Estimate the performance improvement.

Write a method that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the method should return 1367. Incorporate the method into an applet that reads a value from the user. Display the result of the method in the status bar.
6.28 The greatest common divisor (GCD) of two integers is the largest integer that evenly divides each of the two numbers. Write a method gcd that returns the greatest common divisor of two integers. Incorporate the method into an applet that reads two values from the user. Display the result of the method in the status bar.

6.29 Write a method qualityPoints that inputs a student’s average and returns 4 if a student's average is 90–100, 3 if the average is 80–89, 2 if the average is 70–79, 1 if the average is 60–69 and 0 if the average is lower than 60. Incorporate the method into an applet that reads a value from the user. Display the result of the method in the status bar.

6.30 Write an applet that simulates coin tossing. Let the program toss a coin each time the user presses the “Toss” button. Count the number of times each side of the coin appears. Display the results. The program should call a separate method flip that takes no arguments and returns false for tails and true for heads. [Note: If the program realistically simulates coin tossing, each side of the coin should appear approximately half the time.]

6.31 Computers are playing an increasing role in education. Write a program that will help an elementary school student learn multiplication. Use Math.random to produce two positive one-digit integers. The program should then display a question in the status bar, such as

How much is 6 times 7?

The student then types the answer into a JTextField. Next, the program checks the student’s answer. If it is correct, draw the string "Very good!" on the applet and ask another multiplication question. If the answer is wrong, draw the string "No. Please try again." on the applet and let the student try the same question again repeatedly until the student finally gets it right. A separate method should be used to generate each new question. This method should be called once when the applet begins execution and each time the user answers the question correctly. All drawing on the applet should be performed by the paint method.

6.32 The use of computers in education is referred to as computer-assisted instruction (CAI). One problem that develops in CAI environments is student fatigue. This problem can be eliminated by varying the computer’s dialogue to hold the student's attention. Modify the program of Exercise 6.31 so the various comments are printed for each correct answer and each incorrect answer as follows:

Responses to a correct answer

Very good!
Excellent!
Nice work!
Keep up the good work!

Responses to an incorrect answer

No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.

Use random-number generation to choose a number from 1 to 4 that will be used to select an appropriate response to each answer. Use a switch structure in the paint method to issue the responses.

6.33 More sophisticated computer-aided instruction systems monitor the student’s performance over a period of time. The decision to begin a new topic is often based on the student’s success with previous topics. Modify the program of Exercise 6.32 to count the number of correct and incorrect
responses typed by the student. After the student types 10 answers, your program should calculate the percentage of correct responses. If the percentage is lower than 75%, print **Please ask your instructor for extra help** and reset the program so another student can try the program.

**6.34** Write an applet that plays the “guess the number” game as follows: Your program chooses the number to be guessed by selecting a random integer in the range 1 to 1000. The applet displays the prompt **Guess a number between 1 and 1000** next to a **JTextField**. The player types a first guess into the **JTextField** and presses the **Enter** key. If the player’s guess is incorrect, your program should display **Too high. Try again.** or **Too low. Try again.** in the status bar to help the player “zero in” on the correct answer and should clear the **JTextField** so the user can enter the next guess. When the user enters the correct answer, display **Congratulations. You guessed the number!** in the status bar and clear the **JTextField** so the user can play again.  

*Note: The guessing technique employed in this problem is similar to a binary search.*

**6.35** Modify the program of Exercise 6.34 to count the number of guesses the player makes. If the number is 10 or fewer, print **Either you know the secret or you got lucky!** If the player guesses the number in 10 tries, print **Aha! You know the secret!** If the player makes more than 10 guesses, print **You should be able to do better!** Why should it take no more than 10 guesses? Well, with each “good guess” the player should be able to eliminate half of the numbers. Now show why any number from 1 to 1000 can be guessed in 10 or fewer tries.

**6.36** Write a recursive method **power(base, exponent)** that when invoked returns  

\[
\text{base}^\text{exponent}
\]

For example, **power(3, 4)** = \(3 \times 3 \times 3 \times 3\). Assume that **exponent** is an integer greater than or equal to 1.  

(Hint: The recursion step should use the relationship  

\[
\text{base}^\text{exponent} = \text{base} \times \text{base}^{\text{exponent} - 1}
\]

and the terminating condition occurs when **exponent** is equal to 1, because  

\[
\text{base}^1 = \text{base}
\]

Incorporate this method into an applet that enables the user to enter the **base** and **exponent**.

**6.37** (*Towers of Hanoi*) Every budding computer scientist must grapple with certain classic problems, and the **Towers of Hanoi** (see Fig. 6.23) is one of the most famous. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

Let us assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving \(n\) disks can be viewed in terms of moving only \(n - 1\) disks (and hence the recursion) as follows:

a) Move \(n - 1\) disks from peg 1 to peg 2, using peg 3 as a temporary holding area.

b) Move the last disk (the largest) from peg 1 to peg 3.

c) Move the \(n - 1\) disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving \(n - 1\) disk (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area.
Write an applet to solve the Towers of Hanoi problem. Allow the user to enter the number of disks in a `JTextField`. Use a recursive `tower` method with four parameters:

- a) the number of disks to be moved,
- b) the peg on which these disks are initially threaded,
- c) the peg to which this stack of disks is to be moved, and
- d) the peg to be used as a temporary holding area.

Your program should display in a `JTextArea` with scrolling functionality the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

1 \rightarrow 3 (This notation means move one disk from peg 1 to peg 3.)
1 \rightarrow 2
3 \rightarrow 2
1 \rightarrow 3
2 \rightarrow 1
2 \rightarrow 3
1 \rightarrow 3

6.38 Any program that can be implemented recursively can be implemented iteratively, although sometimes with more difficulty and less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version you developed in Exercise 6.37. Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.

6.39 (Visualizing Recursion) It is interesting to watch recursion “in action.” Modify the factorial method of Fig. 6.12 to print its local variable and recursive-call parameter. For each recursive call, display the outputs on a separate line, and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

6.40 The greatest common divisor of integers x and y is the largest integer that evenly divides into both x and y. Write a recursive method `gcd` that returns the greatest common divisor of x and y. The `gcd` of x and y is defined recursively as follows: If y is equal to 0, then `gcd(x, y)` is x; otherwise, `gcd(x, y) = gcd(y, x \% y)`, where `%` is the modulus operator. Use this method to replace the one you wrote in the applet of Exercise 6.28.
6.41 Exercise 6.31 through Exercise 6.33 developed a computer-assisted instruction program to teach an elementary school student multiplication. This exercise suggests enhancements to that program.

a) Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of 2 means to use numbers as large as two digits, etc.

b) Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only and 5 means to intermix randomly problems of all these types.

6.42 Write method distance, to calculate the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\). All numbers and return values should be of type double. Incorporate this method into an applet that enables the user to enter the coordinates of the points.

6.43 What does the following method do?

```java
// Parameter b must be a positive
// integer to prevent infinite recursion
public int mystery( int a, int b )
{
    if ( b == 1 )
        return a;
    else
        return a + mystery( a, b - 1 );
}
```

6.44 After you determine what the program in Exercise 6.43 does, modify the method to operate properly after removing the restriction of the second argument being nonnegative. Also, incorporate the method into an applet that enables the user to enter two integers, and test the method.

6.45 Write an application that tests as many of the math-library methods in Fig. 6.2 as you can. Exercise each of the methods by having your program print out tables of return values for a diversity of argument values.

6.46 Find the error in the following recursive method, and explain how to correct it:

```java
public int sum( int n )
{
    if ( n == 0 )
        return 0;
    else
        return n + sum( n );
}
```

6.47 Modify the craps program of Fig. 6.9 to allow wagering. Initialize variable bankBalance to 1000 dollars. Prompt the player to enter a wager. Check that wager is less than or equal to bankBalance, and if not, have the user reenter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager, and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check if bankBalance has become zero, and if so, print the message "Sorry, You busted!" As the game progresses, print various messages to create some “chatter,” such as "Oh, you’re going for broke, huh?" or "Aw c’mon, take a chance!" or "You’re up big. Now’s the time to cash in your chips!". Implement the “chatter” as a separate method that randomly chooses the string to display.

6.48 Write an applet that uses a method circleArea to prompt the user for the radius of a circle and to calculate and print the area of that circle.
Objectives

- To introduce the array data structure.
- To understand the use of arrays to store, sort and search lists and tables of values.
- To understand how to declare an array, initialize an array and refer to individual elements of an array.
- To be able to pass arrays to methods.
- To understand basic sorting techniques.
- To be able to declare and manipulate multiple-subscript arrays.

With sobs and tears he sorted out
Those of the largest size …
Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing’s so hard, but search will find it out.
Robert Herrick

Now go, write it before them in a table,
and note it in a book.
Isaiah 30:8

‘Tis in my memory lock’d,
And you yourself shall keep the key of it.
William Shakespeare
7.1 Introduction

This chapter serves as an introduction to the important topic of data structures. Arrays are data structures consisting of related data items of the same type. Arrays are “static” entities, in that they remain the same size once they are created, although an array reference may be reassigned to a new array of a different size. Chapter 19, “Data Structures,” introduces dynamic data structures, such as lists, queues, stacks and trees, that can grow and shrink as programs execute. Chapter 20, “Java Utilities Package and Bit Manipulation,” discusses class Vector, which is an array-like class whose objects can grow and shrink in response to a Java program’s changing storage requirements. Chapter 21, “The Collections API,” introduces Java’s predefined data structures that enable the programmer to use existing data structures for lists, queues, stacks and trees rather than “reinventing the wheel.” The Collections API also provides class Arrays, which defines a set of utility methods for array manipulation.
7.2 Arrays

An array is a group of contiguous memory locations that all have the same name and the same type. To refer to a particular location or element in the array, we specify the name of the array and the position number (or index or subscript) of the particular element in the array.

Figure 7.1 shows an integer array called \textit{c}. This array contains 12 elements. A program refers to any one of these elements by giving the name of the array followed by the position number of the particular element in square brackets ([ ]). The first element in every array has position number zero (sometimes called the zeroth element). Thus, the first element of array \textit{c} is \textit{c}[0], the second element of array \textit{c} is \textit{c}[1], the seventh element of array \textit{c} is \textit{c}[6], and, in general, the \textit{i}th element of array \textit{c} is \textit{c}[i - 1]. Array names follow the same conventions as other variable names.

Formally, the position number in square brackets is called a subscript (or an index). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, the program evaluates the expression to determine the subscript. For example, if we assume that variable \textit{a} is 5 and that variable \textit{b} is 6, then the statement

\[
c[ a + b ] += 2;
\]

adds 2 to array element \textit{c}[11]. Note that a subscripted array name is an \textit{lvalue}—it can be used on the left side of an assignment to place a new value into an array element.
Let us examine array \( c \) in Figure 7.1 more closely. The name of the array is \( c \). Every array in Java knows its own length and maintains this information in a variable called \texttt{length}. The expression \( c.length \) accesses array \( c \)’s \texttt{length} variable to determine the length of the array. The array’s 12 elements are referred to as \( c[0], c[1], c[2], \ldots, c[11] \). The value of \( c[0] \) is \(-45\), the value of \( c[1] \) is \( 6 \), the value of \( c[2] \) is \( 0 \), the value of \( c[7] \) is \( 62 \) and the value of \( c[11] \) is \( 78 \). To calculate the sum of the values contained in the first three elements of array \( c \) and store the result in variable \texttt{sum}, we would write

\[
\texttt{sum} = c[0] + c[1] + c[2];
\]

To divide the value of the seventh element of array \( c \) by \( 2 \) and assign the result to the variable \( x \), we would write

\[
\texttt{x} = c[6] / 2;
\]

**Common Programming Error 7.1**

It is important to note the difference between the “seventh element of the array” and “array element seven.” Because array subscripts begin at 0, the “seventh element of the array” has a subscript of 6, while “array element seven” has a subscript of 7 and is actually the eighth element of the array. This confusion is a source of “off-by-one” errors.

The brackets used to enclose the subscript of an array are one of Java’s many operators. Brackets are in the highest level of precedence in Java. The chart in Fig. 7.2 shows the precedence and associativity of the operators introduced so far. They are shown top to bottom in decreasing order of precedence with their associativity and type. See Appendix C for the complete operator precedence chart.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] .</td>
<td>left to right</td>
<td>highest</td>
</tr>
<tr>
<td>++ --</td>
<td>right to left</td>
<td>unary postfix</td>
</tr>
<tr>
<td>++ -- + + !</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
<td>boolean logical AND</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
<td>boolean logical exclusive OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>= + = - = * = /= % =</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

Fig. 7.2 Precedence and associativity of the operators discussed so far.
7.3 Declaring and Allocating Arrays

Arrays are objects that occupy space in memory. All objects in Java (including arrays) must be allocated dynamically with operator `new`. For an array, the programmer specifies the type of the array elements and the number of elements as part of the `new` operation. To allocate 12 elements for integer array `c` in Fig. 7.1, use the declaration:

```java
int c[] = new int[12];
```

The preceding declaration also can be performed in two steps as follows:

```java
int c[];            // declares the array
c = new int[12];   // allocates the array
```

When allocating an array, each element of the array receives a default value—zero for the numeric primitive-data-type elements, `false` for `boolean` elements or `null` for references (any nonprimitive type).

**Common Programming Error 7.2**

Unlike array declarations in several other programming languages (such as C and C++), Java array declarations must not specify the number of array elements in the square brackets after the array name; otherwise, a syntax error occurs. For example, the declaration:

```java
int c[12];
```

causes a syntax error.

A program can allocate memory for several arrays with a single declaration. The following declaration reserves 100 elements for `String` array `b` and 27 elements for `String` array `x`:

```java
String b[] = new String[100], x[] = new String[27];
```

When declaring an array, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all identifiers in the declaration represent arrays, as in

```java
double[] array1, array2;
```

which declares both `array1` and `array2` as arrays of `double` values. As shown previously, the declaration and initialization of the array can be combined in the declaration. The following declaration reserves 10 elements for `array1` and 20 elements for `array2`:

```java
double[] array1 = new double[10],
array2 = new double[20];
```

A program can declare arrays of any data type. It is important to remember that every element of a primitive data type array contains one value of the declared data type. For example, every element of an `int` array contains an `int` value. However, in an array of a nonprimitive type, every element of the array is a reference to an object of the array’s declared data type. For example, every element of a `String` array is a reference to a `String`. In arrays that store references, the references have the value `null` by default.

7.4 Examples Using Arrays

This section presents several examples using arrays that demonstrate declaring arrays, allocating arrays, initializing arrays and manipulating array elements in various ways. For
simplicity, the examples in this section use arrays that contain elements of type \texttt{int}. Please remember that a program can declare an array of any data type.

### 7.4.1 Allocating an Array and Initializing Its Elements

The application of Figure 7.3 uses operator \texttt{new} to allocate dynamically an array of 10 \texttt{int} elements, which are initially zero (the default value in an array of type \texttt{int}). The program displays the array elements in tabular format in a \texttt{JTextArea}.

Line 12 declares \texttt{array}—a reference capable of referring to an array of \texttt{int} elements. Line 14 allocates the 10 elements of the array with \texttt{new} and initializes the reference. The program builds its output in the \texttt{String} called \texttt{output} that will be displayed in a \texttt{JTextArea} on a message dialog. Line 16 appends to \texttt{output} the headings for the columns displayed by the program. The columns represent the subscript for each array element and the value of each array element, respectively.

Lines 19–20 use a \texttt{for} structure to append the subscript number (represented by \texttt{counter}) and value of each array element to \texttt{output}. Note the use of zero-based counting (remember, subscripts start at 0), so that the loop can access every array element. Also, note the expression \texttt{array.length} in the \texttt{for} structure condition to determine the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable \texttt{counter} is less than 10. For a 10-element array, the subscript values are 0 through 9, so using the less than operator < guarantees that the loop does not attempt to access an element beyond the end of the array.

```java
// Fig. 7.3: InitArray.java
// Creating an array.
import javax.swing.*;

public class InitArray {
    // main method begins execution of Java application
    public static void main( String args[] )
    {
        int array[];       // declare reference to an array

        array = new int[ 10 ];  // dynamically allocate array

        String output = "Subscript\tValue\n";

        // append each array element's value to String output
        for ( int counter = 0; counter < array.length; counter++ )
            output += counter + "\t" + array[ counter ] + "\n";

        JTextArea outputArea = new JTextArea();
        outputArea.setText( output );
    }
}
```

Fig. 7.3 Initializing the elements of an array to zeros (part 1 of 2).
7.4.2 Using an Initializer List to Initialize Elements of an Array

A program can allocate and initialize the elements of an array in the array declaration by following the declaration with an equal sign and a comma-separated initializer list enclosed in braces ({ and }). In this case, the array size is determined by the number of elements in the initializer list. For example, the declaration

```java
int n[] = { 10, 20, 30, 40, 50 };
```

creates a five-element array with subscripts of \(0, 1, 2, 3\) and \(4\). Note that the preceding declaration does not require the `new` operator to create the array object. When the compiler encounters an array declaration that includes an initializer list, the compiler counts the number of initializers in the list and sets up a new operation to allocate the appropriate number of array elements.

The application of Fig. 7.4 initializes an integer array with 10 values (line 14) and displays the array in tabular format in a `JTextArea` on a message dialog. The code for displaying the array elements is identical to that of Fig. 7.3.

```java
25 JOptionPane.showMessageDialog( null, outputArea,
26 "Initializing an Array of int Values",
27 JOptionPane.INFORMATION_MESSAGE );
28 System.exit( 0 );
30 }
31 }
```

**Fig. 7.3** Initializing the elements of an array to zeros (part 2 of 2).

**Fig. 7.4** Initializing the elements of an array with a declaration (part 1 of 2).
7.4.3 Calculating the Value to Store in Each Array Element

Some programs calculate the value stored in each array element. The application of Fig. 7.5 initializes the elements of a 10-element array to the even integers from 2 to 20 (2, 4, 6, ..., 20) and displays the array in tabular format. The for structure at lines 18–19 generates an array element’s value by multiplying the current value of counter (the control variable of the loop) by 2 and adding 2.
// Fig. 7.5: InitArray.java
// Initialize array with the even integers from 2 to 20.

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

public class InitArray {
    // main method begins execution of Java application
    public static void main( String args[] ) {
        final int ARRAY_SIZE = 10;            // reference to int array
        int array[];                         // allocate array
        array = new int[ ARRAY_SIZE ];       // allocate array

        // calculate value for each array element
        for ( int counter = 0; counter < array.length; counter++ )
            array[ counter ] = 2 + 2 * counter;

        String output = "Subscript\tValue\n";
        for ( int counter = 0; counter < array.length; counter++ )
            output += counter + "\t" + array[ counter ] + "\n";

        JTextArea outputArea = new JTextArea();
        outputArea.setText( output );

        JOptionPane.showMessageDialog( null, outputArea,
            "Initializing to Even Numbers from 2 to 20",
            JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }
}
Line 12 uses the `final` qualifier to declare constant variable `ARRAY_SIZE`, whose value is `10`. Remember that constant variables must be initialized before they are used and cannot be modified thereafter. If an attempt is made to modify a `final` variable after it is declared as shown on line 12, the compiler issues a message like

```c
cannot assign a value to final variable variableName
```

If an attempt is made to use a `final` variable before it is initialized, the compiler issues the error message

```c
Variable variableName may not have been initialized
```

Constant variables also are called named constants or read-only variables. Such variables often can make programs more readable. Note that the term “constant variable” is an oxymoron—a contradiction in terms—like “jumbo shrimp” or “freezer burn.”

**Common Programming Error 7.3**

Assigning a value to a constant variable after the variable has been initialized is a syntax error.

### 7.4.4 Summing the Elements of an Array

Often, the elements of an array represent a series of values to use in a calculation. For example, if the elements of an array represent the grades for an exam in a class, the professor may wish to total the elements of an array, then calculate the class average for the exam.

The application of Fig. 7.6 sums the values contained in the 10-element integer array. The program declares, allocates and initializes the array at line 12. The `for` structure at lines 16–17 performs the calculations. [Note: It is important to remember that the values being supplied as initializers for `array` normally would be read into the program. For example, in an applet the user could enter the values through a `JTextField`, or in an application the values could be read from a file on disk (discussed in Chapter 16). Reading the data into a program makes the program more flexible, because it can be used with different sets of data.]

```java
// Fig. 7.6: SumArray.java
// Total the values of the elements of an array.

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

public class SumArray {
    // main method begins execution of Java application
    public static void main( String args[] )
    {
        int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // Fig. 7.6
        int total = 0;
    }
}
```

Fig. 7.6  Computing the sum of the elements of an array (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
7.4.5 Using Histograms to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent larger numeric values. One simple way to display numeric data graphically is with a histogram that shows each numeric value as a bar of asterisks (*).

Our next application (Fig. 7.7) reads numbers from an array and graphs the information in the form of a bar chart (or histogram). The program displays each number followed by a bar consisting of that many asterisks. The nested for loop (lines 17–24) appends the bars to the String that will be displayed in JTextArea outputArea on a message dialog. Note the loop continuation condition of the inner for structure at line 22 (stars <= array[counter]). Each time the program reaches the inner for structure, the loop counts from 1 to array[counter], thus using a value in array to determine the final value of the control variable stars and the number of asterisks to display.

```java
15    // add each element's value to total
16    for ( int counter = 0; counter < array.length; counter++ )
17        total += array[counter];
18
19    JOptionPane.showMessageDialog( null,
20        "Total of array elements: " + total,
21        "Sum the Elements of an Array",
22        JOptionPane.INFORMATION_MESSAGE );
23
24    System.exit( 0 );
25 }
26 }
```

Fig. 7.6 Computing the sum of the elements of an array (part 2 of 2).

Fig. 7.7 A program that prints histograms (part 1 of 2).
7.4.6 Using the Elements of an Array as Counters

Sometimes programs use a series of counter variables to summarize data, such as the results of a survey. In Chapter 6, we used a series of counters in our die-rolling program to track the number of occurrences of each side on a six-sided die as the program rolled the die 6000 times. We also indicated that there is a more elegant method of writing the program of Fig. 6.8. An array version of this application is shown in Fig. 7.8.

```java
// for each array element, output a bar in histogram
for ( int counter = 0; counter < array.length; counter++ ) {
    output +=
        "\n" + counter + "\t" + array[ counter ] + "\t";
    // print bar of asterisks
    for ( int stars = 0; stars < array[ counter ]; stars++ )
        output += ";
}
```

```
JTextArea outputArea = new JTextArea();
outputArea.setText( output );
JOptionPane.showMessageDialog( null, outputArea,
        "Histogram Printing Program",
        JOptionPane.INFORMATION_MESSAGE );
System.exit( 0 );
```
The program uses the seven-element array `frequency` to count the occurrences of each side of the die. This program replaces lines 20–46 of Fig. 6.8 with line 19 of this program. Line 19 uses the random `face` value as the subscript for array `frequency` to determine which element the program should increment during each iteration of the loop. Since the random number calculation on line 19 produces numbers from 1 to 6 (the values for a six-sided die), the `frequency` array must be large enough to store six counters. However, in this program, we chose to use a seven-element array. We ignore the first array element, `frequency[0]`, because it is more logical to have the face value 1 increment...
frequency[1] than frequency[0]. This allows us to use each face value directly as a subscript for the frequency array.

**Good Programming Practice 7.1**

Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

**Performance Tip 7.1**

Sometimes performance considerations far outweigh clarity considerations.

Also, lines 25–26 of this program replace lines 52–55 from Fig. 6.8. We can loop through array frequency, so we do not have to enumerate each line of text to display in the JTextArea as we did in Fig. 6.8.

### 7.4.7 Using Arrays to Analyze Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the problem statement:

*Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 means awful and 10 means excellent). Place the 40 responses in an integer array and summarize the results of the poll.*

This is a typical array processing application (see Fig. 7.9). We wish to summarize the number of responses of each type (i.e., 1 through 10). The array responses is a 40-element integer array of the students’ responses to the survey. We use an 11-element array frequency to count the number of occurrences of each response. As in Fig. 7.8, we ignore the first element (frequency[0]) because it is more logical to have the response 1 increment frequency[1] than frequency[0]. This allows us to use each response directly as a subscript on the frequency array. Each element of the array is used as a counter for one of the survey responses.

```
1 // Fig. 7.9: StudentPoll.java
2 // Student poll program
3 // Java extension packages
4 import javax.swing.*;
5
6 public class StudentPoll {
7     // main method begins execution of Java application
8     public static void main( String args[] )
9     {
10         int responses[] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
11             1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
12             6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
13             5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
14         int frequency[] = new int[ 11 ];
```

*Fig. 7.9* A simple student-poll analysis program (part 1 of 2).
The \texttt{for} loop (lines 21–22) takes the responses one at a time from array \texttt{responses} and increments one of the 10 counters in the \texttt{frequency} array (\texttt{frequency[1]} to \texttt{frequency[10]}). The key statement in the loop is line 22, which increments the appropriate \texttt{frequency} counter, depending on the value of \texttt{responses[answer]}.

Let’s consider several iterations of the \texttt{for} loop. When counter \texttt{answer} is 0, \texttt{responses[answer]} is the value of the first element of array \texttt{responses} (i.e., 1), so the program interprets \texttt{++frequency[responses[answer]]}; as

\begin{verbatim}
  ++frequency[1];
\end{verbatim}

which increments the value in array element one. To evaluate the expression, start with the value in the innermost set of square brackets (\texttt{answer}). Once you know the value of \texttt{answer}, plug that value into the expression and evaluate the next outer set of square brackets.
(responses[answer]). Then, use that value as the subscript for the frequency array to determine which counter to increment.

When answer is 1, responses[answer] is the value of responses’ second element (2), so the program interprets ++frequency[responses[answer]]; as

```
++frequency[ 2 ];
```

which increments array element two (the third element of the array).

When answer is 2, responses[answer] is the value of responses’ third element (6), so the program interprets ++frequency[responses[answer]]; as

```
++frequency[ 6 ];
```

which increments array element six (the seventh element of the array), and so on. Regardless of the number of responses processed in the survey, the program requires only an 11-element array (ignoring element zero) to summarize the results, because all the response values are between 1 and 10 and the subscript values for an 11-element array are 0 through 10. Also, note that the summarized results are correct, because the elements of array frequency were initialized to zero when the array was allocated with new.

If the data contained invalid values, such as 13, the program would attempt to add 1 to frequency[13]. This is outside the bounds of the array. In the C and C++ programming languages, such a reference would be allowed by the compiler and at execution time. The program would “walk” past the end of the array to where it thought element number 13 was located and add 1 to whatever happened to be at that location in memory. This could potentially modify another variable in the program or even result in premature program termination. Java provides mechanisms to prevent accessing elements outside the bounds of the array.

**Common Programming Error 7.4**

Referring to an element outside the array bounds is a logic error.

**Testing and Debugging Tip 7.1**

When a Java program executes, the Java interpreter checks array element subscripts to be sure they are valid (i.e., all subscripts must be greater than or equal to 0 and less than the length of the array). If there is an invalid subscript, Java generates an exception.

**Testing and Debugging Tip 7.2**

Exceptions indicate that an error occurred in a program. A programmer can write code to recover from an exception and continue program execution instead of abnormally terminating the program. When an invalid array reference occurs, Java generates an ArrayIndexOutOfBoundsException. Chapter 14 covers exception handling in detail.

**Testing and Debugging Tip 7.3**

When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (one less than the size of the array). The loop terminating condition should prevent accessing elements outside this range.

**Testing and Debugging Tip 7.4**

Programs should validate all input values to prevent erroneous information from affecting a program’s calculations.
7.5 References and Reference Parameters

Two ways to pass arguments to methods (or functions) in many programming languages (like C and C++) are pass-by-value and pass-by-reference (also called call-by-value and call-by-reference). When an argument is passed by value, a copy of the argument’s value is made and passed to the called method.

**Testing and Debugging Tip 7.5**

With pass-by-value, changes to the called method’s copy do not affect the original variable’s value in the calling method. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.

When an argument is passed by reference, the caller gives the called method the ability to access the caller’s data directly and to modify that data if the called method so chooses. Pass-by-reference improves performance, because it eliminates the overhead of copying large amounts of data.

**Software Engineering Observation 7.1**

Unlike other languages, Java does not allow the programmer to choose whether to pass each argument by value or by reference. Primitive data type variables are always passed by value. Objects are not passed to methods; rather, references to objects are passed to methods. The references themselves are passed by value—a copy of a reference is passed to a method. When a method receives a reference to an object, the method can manipulate the object directly.

**Software Engineering Observation 7.2**

When returning information from a method via a return statement, primitive-data-type variables are always returned by value (i.e., a copy is returned) and objects are always returned by reference (i.e., a reference to the object is returned).

To pass a reference to an object into a method, simply specify the reference name in the method call. Mentioning the reference by its parameter name in the body of the called method actually refers to the original object in memory, and the original object can be accessed directly by the called method.

Arrays are treated as objects by Java; therefore, arrays are passed to methods by reference—a called method can access the elements of the caller’s original arrays. The name of an array is actually a reference to an object that contains the array elements and the length instance variable, which indicates the number of elements in the array. In the next section, we demonstrate pass-by-value and pass-by-reference using arrays.

**Performance Tip 7.2**

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would waste time and would consume considerable storage for the copies of the arrays.

7.6 Passing Arrays to Methods

To pass an array argument to a method, specify the name of the array without any brackets. For example, if array `hourlyTemperatures` is declared as

```
int hourlyTemperatures[] = new int[24];
```

the method call

    modifyArray( hourlyTemperatures );

passes array `hourlyTemperatures` to method `modifyArray`. In Java, every array object “knows” its own size (via the `length` instance variable). Thus, when we pass an array object into a method, we are not required to pass the size of the array as an additional argument.

Although entire arrays and objects referred to by individual elements of a nonprimitive-type array are passed by reference, individual array elements of primitive data types are passed by value exactly as simple variables are. Such simple single pieces of data are called `scalars` or `scalar quantities`. To pass an array element to a method, use the subscripted name of the array element as an argument in the method call.

For a method to receive an array through a method call, the method’s parameter list must specify an array parameter (or several if more than one array is to be received). For example, the method header for method `modifyArray` might be written as

    void modifyArray( int b[] )

indicating that `modifyArray` expects to receive an integer array in parameter `b`. Since arrays are passed by reference, when the called method uses the array name `b`, it refers to the actual array in the caller (array `hourlyTemperatures` in the preceding call).

The applet of Fig. 7.10 demonstrates the difference between passing an entire array and passing an array element. Once again, we use an applet here, because we have not yet defined an application that contains methods other than `main`. We are still taking advantage of some of the features provided for free in an applet (such as the automatic creation of an applet object and the automatic calls to `init`, `start` and `paint` by the applet container). In Chapter 9, Object-Oriented Programming, we introduce applications that execute in their own windows. At that point, we will begin to see application classes containing several methods.

Lines 15–17 in method `init` define the `JTextArea` called `outputArea` and attach it to the applet’s content pane. The `for` structure at lines 26–27 appends the five elements of `array` (an array of `int` values) to the `String` called `output`. Line 29 invokes method `modifyArray` and passes it `array` as an argument. Method `modifyArray` (lines 50–54) multiplies each element by 2. To illustrate that `array`’s elements were modified, the `for` structure at lines 34–35 appends the five elements of `array` to `output` again. As the screen capture shows, method `modifyArray` did change the value of each element.

```
1 // Fig. 7.10: PassArray.java
2 // Passing arrays and individual array elements to methods
3
4 // Java core packages
5 import java.awt.Container;
6
7 // Java extension packages
8 import javax.swing.*;

Fig. 7.10 Passing arrays and individual array elements to methods (part 1 of 3).
© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
```
public class PassArray extends JApplet {

    // initialize applet
    public void init()
    {
        JTextArea outputArea = new JTextArea();
        Container container = getContentPane();
        container.add(outputArea);

        int array[] = { 1, 2, 3, 4, 5 };

        String output = "Effects of passing entire array by reference:
        The values of the original array are:
        
        // append original array elements to String output
        for (int counter = 0; counter < array.length; counter++ )
        
            output += "   " + array[counter];

        modifyArray( array ); // array passed by reference

        output += "\n\nThe values of the modified array are:
"

        // append modified array elements to String output
        for (int counter = 0; counter < array.length; counter++ )

            output += "   " + array[counter];

        output += "\n\nEffects of passing array " +
"element by value:\n" +
"a[3] before modifyElement: " + array[3];

        // attempt to modify array[3]
        modifyElement( array[3] );

        output += "\na[3] after modifyElement: " + array[3];

        outputArea.setText(output);
    }

    // multiply each element of an array by 2
    public void modifyArray( int array2[] )
    {
        for (int counter = 0; counter < array2.length; counter++ )
            array2[counter] *= 2;
    }

    // multiply argument by 2
    public void modifyElement( int element )
    {
        element *= 2;
    }

} // end class PassArray

Fig. 7.10 Passing arrays and individual array elements to methods (part 2 of 3).
Next, the program demonstrates that individual elements of primitive-type arrays are passed to methods by value. To show the value of `array[3]` before calling method `modifyElement`, lines 37–39 append the value of `array[3]` (and other information) to `output`. Line 42 invokes method `modifyElement` and passes `array[3]` as an argument. Remember that `array[3]` is actually one `int` value in `array`. Also, remember that values of primitive types are passed to methods by value. Therefore, the program passes a copy of `array[3]`. Method `modifyElement` multiplies its argument by 2 and stores the result in its parameter `element`. Method parameters are local variables, so when `modifyElement` terminates, the local variable `element` is destroyed. Thus, when the program returns control to `init`, line 44 appends the unmodified value of `array[3]` `output`. Line 45 displays the results in the `JTextArea`.

### 7.7 Sorting Arrays

Sorting data (i.e., placing the data into some particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, within that, by first name, to make it easy to find phone numbers. Virtually every organization must sort some data and in many cases massive amounts of data. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science. In this chapter, we discuss one of the simplest sorting schemes. In the exercises in this chapter, Chapter 19 and Chapter 21, we investigate more complex schemes that yield superior performance.

**Performance Tip 7.3**

Sometimes, the simplest algorithms perform poorly. Their virtue is that they are easy to program, test and debug. Sometimes, more complex algorithms are required to realize maximum performance.

Figure 7.11 sorts the values of `array` (a the 10-element array of `int` values) into ascending order. The technique we use is called the `bubble sort` or the `sinking sort`, because the smaller values gradually “bubble” their way to the top of the array (i.e., toward the first element) like air bubbles rising in water, while the larger values sink to the bottom (end) of the array. The technique uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values
are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the array. The applet contains methods init, bubbleSort and swap. Method init (lines 13–36) initializes the applet. Method bubbleSort (lines 39–58) is called from init to sort the elements of array. Method bubbleSort calls method swap (lines 61–68) as necessary to exchange two elements of the array.

Lines 24–25 append the original values of array to the String called output. Line 27 invokes method bubbleSort and passes array as the array to sort.

Method bubbleSort receives the array as parameter array2. The nested for structure at lines 42–56 performs the sort. The outer loop controls the number of passes of the array. The inner loop controls the comparisons and swapping (if necessary) of the elements during each pass.
Method \texttt{bubbleSort} first compares \texttt{array2[0]} to \texttt{array2[1]}, then \texttt{array2[1]} to \texttt{array2[2]}, then \texttt{array2[2]} to \texttt{array2[3]} so on until it completes the pass by comparing \texttt{array2[8]} to \texttt{array2[9]}. Although there are 10 elements, the comparison loop performs only nine comparisons. The comparisons performed in a bubble sort could cause a large value to move down the array (sink) many positions on a single pass. However, a small value may move up (bubble) only one position per pass. On the first pass, the largest value is guaranteed to sink to the bottom element of the array,
array2[9]. On the second pass, the second largest value is guaranteed to sink to array2[8]. On the ninth pass, the ninth largest value sinks to array2[1]. This leaves the smallest value in array2[0], so only nine passes are required to sort a 10-element array.

If a comparison reveals that the two elements are in descending order, bubbleSort calls method swap to exchange the two elements, so they will be in ascending order in the array. Method swap receives a reference to the array (which it calls array3) and two integers representing the subscripts of the two elements of the array to exchange. The exchange is performed by the three assignments

```java
hold = array3[first];
array3[first] = array3[second];
array3[second] = hold;
```

where the extra variable hold temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```java
array3[first] = array3[second];
array3[second] = array3[first];
```

If array3[first] is 7 and array3[second] is 5, after the first assignment both array elements contain 5 and the value 7 is lost, hence the need for the extra variable hold.

The chief virtue of the bubble sort is that it is easy to program. However, the bubble sort runs slowly. This becomes apparent when sorting large arrays. In Exercise 7.11, we ask you to develop more efficient versions of the bubble sort. Other exercises investigate some sorting algorithms that are far more efficient than the bubble sort. More advanced courses (often titled “Data Structures,” “Algorithms” or “Computational Complexity”) investigate sorting and searching in greater depth.

### 7.8 Searching Arrays: Linear Search and Binary Search

Often, a programmer will be working with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain key value. The process of locating a particular element value in an array is called searching. In this section, we discuss two searching techniques—the simple linear search technique and the more efficient binary search technique. Exercise 7.31 and Exercise 7.32 at the end of this chapter ask you to implement recursive versions of the linear search and the binary search.

#### 7.8.1 Searching an Array with Linear Search

In the applet of Fig. 7.12, method linearSearch (defined at lines 52–62) uses a for structure (lines 55–59) containing an if structure to compare each element of an array with a search key. If the search key is found, the method returns the subscript value for the element to indicate the exact position of the search key in the array. If the search key is not found, the method returns -1 to indicate that the search key was not found. We return -1 because it is not a valid subscript number. If the array being searched is not in any particular order, it is just as likely that the search key will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array.
Figure 7.12 contains a 100-element array filled with the even integers from 0 to 198. The user types the search key in a JTextField and presses Enter to start the search. [Note: We pass the array to linearSearch even though the array is an instance variable of the class. We do this because an array normally is passed to a method of another class for sorting. For example, class Arrays (see Chapter 21) contains a variety of static methods for sorting arrays, searching arrays, comparing the contents of arrays and filling arrays of all the primitive types, Objects and Strings.]

```java
// Fig. 7.12: LinearSearch.java
// Linear search of an array

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

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

// Fig. 7.12: LinearSearch.java
// Linear search of an array

public class LinearSearch extends JApplet
    implements ActionListener {

    JLabel enterLabel, resultLabel;
    JTextField enterField, resultField;
    int array[];

    // set up applet's GUI
    public void init() {
        // get content pane and set its layout to FlowLayout
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // set up JLabel and JTextField for user input
        enterLabel = new JLabel( "Enter integer search key" );
        enterLabel.setHorizontalAlignment( JLabel.CENTER );
        container.add( enterLabel );

        enterField = new JTextField( 10 );
        container.add( enterField );

        // register this applet as enterField's action listener
        enterField.addActionListener( this );

        // set up JLabel and JTextField for displaying results
        resultLabel = new JLabel( "Result" );
        container.add( resultLabel );

        resultField = new JTextField( 20 );
        resultField.setEditable( false );
        container.add( resultField );

        // create array and populate with even integers 0 to 198
        array = new int[ 100 ];
    }
}
```

Fig. 7.12  Linear search of an array (part 1 of 2).
The linear search method works well for small arrays or for unsorted arrays. However, for large arrays, linear searching is inefficient. If the array is sorted, the high-speed \textit{binary search} technique presented in the next section can be used.

\begin{figure}
\includegraphics[width=\textwidth]{fig7_12}
\caption{Linear search of an array (part 2 of 2).}
\end{figure}
7.8.2 Searching a Sorted Array with Binary Search

The binary search algorithm eliminates half of the elements in the array being searched after each comparison. The algorithm locates the middle array element and compares it to the search key. If they are equal, the search key has been found and binary search returns the subscript of that element. Otherwise, binary search reduces the problem to searching half of the array. If the search key is less than the middle array element, the first half of the array will be searched; otherwise, the second half of the array will be searched. If the search key is not the middle element in the specified subarray (piece of the original array), the algorithm repeats on one quarter of the original array. The search continues until the search key is equal to the middle element of a subarray or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).

In a worst-case scenario, searching a sorted array of 1024 elements will take only 10 comparisons using a binary search. Repeatedly dividing 1024 by 2 (because after each comparison we are able to eliminate half of the array) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 \( (2^{10}) \) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 \( (2^{20}) \) elements takes a maximum of 20 comparisons to find the key. An array of one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous increase in performance over the linear search that required comparing the search key to an average of half the elements in the array. For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

Figure 7.13 presents the iterative version of method `binarySearch` (lines 85–116). The method receives two arguments—an integer array called `array2` (the array to search) and an integer `key` (the search key). The program passes the array to `binarySearch` even though the array is an instance variable of the class. Once again, we do this because an array normally is passed to a method of another class for sorting. If `key` matches the `middle` element of a subarray, `binarySearch` returns `middle` (the subscript of the current element) to indicate that the value was found and the search is complete. If `key` does not match the `middle` element of a subarray, `binarySearch` adjusts the `low` subscript or `high` subscript (both declared in the method), to continue the search using a smaller subarray. If `key` is less than the middle element, the `high` subscript is set to `middle - 1` and the search continues on the elements from `low` to `middle - 1`. If `key` is greater than the middle element, the `low` subscript is set to `middle + 1` and the search continues on the elements from `middle + 1` to `high`. Method `binarySearch` performs these comparisons in the nested `if/else` structure at lines 102–111.

```java
1 // Fig. 7.13: BinarySearch.java
2 // Binary search of a sorted array
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.text.*;
```

Fig. 7.13  Binary search of a sorted array (part 1 of 5)
import javax.swing.*;

public class BinarySearch extends JApplet implements ActionListener {

    JLabel enterLabel, resultLabel;
    JTextField enterField, resultField;
    JTextArea output;

    int array[];
    String display = "";

    // set up applet's GUI
    public void init() {
        // get content pane and set its layout to FlowLayout
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // set up JLabel and JTextField for user input
        enterLabel = new JLabel( "Enter integer search key" );
        container.add( enterLabel );
        enterField = new JTextField( 10 );
        container.add( enterField );
        enterField.addActionListener( this );

        // set up JLabel and JTextField for displaying results
        resultLabel = new JLabel( "Result" );
        container.add( resultLabel );
        resultField = new JTextField( 20 );
        resultField.setEditable( false );
        container.add( resultField );

        // set up JTextArea for displaying comparison data
        output = new JTextArea( 6, 60 );
        output.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
        container.add( output );

        // create array and fill with even integers 0 to 28
        array = new int[ 15 ];

        for ( int counter = 0; counter < array.length; counter++ )
            array[ counter ] = 2 * counter;

    } // end method init
```java
public void actionPerformed( ActionEvent actionEvent )
{
    // input also can be obtained with enterField.getText()
    String searchKey = actionEvent.getActionCommand();

    // initialize display string for new search
    display = "Portions of array searched\n";

    // perform binary search
    int element =
        binarySearch( array, Integer.parseInt( searchKey ) );

    output.setText( display );

    // display search result
    if ( element != -1 )
        resultField.setText( "Found value in element " + element );
    else
        resultField.setText( "Value not found" );
}

// method to perform binary search of an array
public int binarySearch( int array2[], int key )
{
    int low = 0;                  // low element subscript
    int high = array.length - 1;  // high element subscript
    int middle;                   // middle element subscript

    // loop until low subscript is greater than high subscript
    while ( low <= high ) {
        // determine middle element subscript
        middle = ( low + high ) / 2;

        // display subset of array elements used in this
        // iteration of binary search loop
        buildOutput( array2, low, middle, high );

        // if key matches middle element, return middle location
        if ( key == array[ middle ] )
            return middle;

        // if key less than middle element, set new high element
        else if ( key < array[ middle ] )
            high = middle - 1;

        // key greater than middle element, set new low element
        else
            low = middle + 1;
    }
}
```

Fig. 7.13 Binary search of a sorted array (part 3 of 5)
return -1;  // key not found

// build row of output showing subset of array elements
currently being processed
void buildOutput( int array3[], int low, int middle, int high )
{
   // create 2-digit integer number format
   DecimalFormat twoDigits = new DecimalFormat( "00" );

   // loop through array elements
   for ( int counter = 0; counter < array3.length; counter++ ) {
      if ( counter < low || counter > high )
         display += "    ";

      // if middle element, append element to String display
      // followed by asterisk (*) to indicate middle element
      else if ( counter == middle )
         display +=
            twoDigits.format( array3[ counter ] ) + "* ";

      else
         display +=
            twoDigits.format( array3[ counter ] ) + "  ";

   }  // end for structure

   display += "\n";

}  // end method buildOutput

} // end class BinarySearch

Fig. 7.13  Binary search of a sorted array (part 4 of 5)
The program uses a 15-element array. The first power of 2 greater than the number of array elements is 16 \( (2^4) \); therefore, `binarySearch` requires at most four comparisons to find the `key`. To illustrate this, line 99 of method `binarySearch` calls method `buildOutput` (defined at lines 120–150) to output each subarray during the binary search process. Method `buildOutput` marks the middle element in each subarray with an asterisk (*) to indicate the element to which the `key` is compared. Each search in this example results in a maximum of four lines of output—one per comparison.

`JTextArea output` uses `Monospaced` font (a fixed-width font—i.e., all characters are the same width) to help align the displayed text in each line of output. Line 49 uses method `setFont` to change the font displayed in `output`. Method `setFont` can change the font of text displayed on most GUI components. The method requires a `Font` (package `java.awt`) object as its argument. A `Font` object is initialized with three arguments—the `String` name of the font ("Monospaced"), an `int` representing the style of the font (`Font.PLAIN` is a constant integer defined in class `Font` that indicates plain font) and an `int` representing the point size of the font (12). Java provides generic names for several fonts available on every Java platform. `Monospaced` font is also called `Courier`. Other common fonts include `Serif` (also called `TimesRoman`) and `SansSerif` (also called `Helvetica`). Java 2 actually provides access to all fonts on your system through methods of class `GraphicsEnvironment`. The style can also be `Font.BOLD`, `Font.ITALIC` or `Font.BOLD + Font.ITALIC`. The point size represents the size of the font—there are 72 points to an inch. The actual size of the text as it appears on the screen may vary based on the size of the screen and the screen resolution. We discuss font manipulation again in Chapter 11.
Multiple-Subscripted Arrays

Multiple-subscripted arrays with two subscripts are often used to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify the two subscripts—by convention, the first identifies the element’s row and the second identifies the element’s column. Arrays that require two subscripts to identify a particular element are called double-subscripted arrays or two-dimensional arrays. Note that multiple-subscripted arrays can have more than two subscripts. Java does not support multiple-subscripted arrays directly, but does allow the programmer to specify single-subscripted arrays whose elements are also single-subscripted arrays, thus achieving the same effect. Figure 7.14 illustrates a double-subscripted array, \( a \), containing three rows and four columns (i.e., a 3-by-4 array). In general, an array with \( m \) rows and \( n \) columns is called an \( m \text{-by-} n \) array.

Every element in array \( a \) is identified in Fig. 7.14 by an element name of the form \( a[\text{row}][\text{column}] \); \( a \) is the name of the array and \( \text{row} \) and \( \text{column} \) are the subscripts that uniquely identify the row and column of each element in \( a \). Notice that the names of the elements in the first row all have a first subscript of \( 0 \); the names of the elements in the fourth column all have a second subscript of \( 3 \).

Multiple-subscripted arrays can be initialized with initializer lists in declarations like a single-subscripted array. A double-subscripted array \( b[2][2] \) could be declared and initialized with

```java
int b[][] = { { 1, 2 }, { 3, 4 } };
```

The values are grouped by row in braces. So, \( 1 \) and \( 2 \) initialize \( b[0][0] \) and \( b[0][1] \), and \( 3 \) and \( 4 \) initialize \( b[1][0] \) and \( b[1][1] \). The compiler determines the number of rows by counting the number of initializer sublists (represented by sets of braces) in the initializer list. The compiler determines the number of columns in each row by counting the number of initializer values in the initializer sublist for that row.

Multiple-subscripted arrays are maintained as arrays of arrays. The declaration

```java
int b[][] = { { 1, 2 }, { 3, 4, 5 } };
```

Figure 7.14 A double-subscripted array with three rows and four columns.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
creates integer array \( b \) with row 0 containing two elements (1 and 2) and row 1 containing three elements (3, 4 and 5).

A multiple-subscripted array with the same number of columns in every row can be allocated dynamically. For example, a 3-by-4 array is allocated as follows:

\[
\text{int } b[\ast]\[\ast]; \\
b = \text{new int[ 3 ][ 4 ];}
\]

In this case, we the literal values 3 and 4 to specify the number of rows and number of columns, respectively. Note that programs also can use variables to specify array dimensions. As with single-subscripted arrays, the elements of a double-subscripted array are initialized when \texttt{new} creates the array object.

A multiple-subscripted array in which each row has a different number of columns can be allocated dynamically as follows:

\[
\text{int } b[\ast]\[\ast]; \\
b = \text{new int[ 2 ][ ];} \quad \text{ // allocate rows} \\
b[0] = \text{new int[ 5 ];} \quad \text{ // allocate columns for row } 0 \\
b[1] = \text{new int[ 3 ];} \quad \text{ // allocate columns for row } 1
\]

The preceding code creates a two-dimensional array with two rows. Row 0 has five columns and row 1 has three columns.

The applet of Fig. 7.15 demonstrates initializing double-subscripted arrays in declarations and using nested \texttt{for} loops to traverse the arrays (i.e., manipulate every element of the array).

```java
// Fig. 7.15: InitArray.java
// Initializing multidimensional arrays

// Java core packages
import java.awt.Container;

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

public class InitArray extends JApplet {
    JTextArea outputArea;
    
    // set up GUI and initialize applet
    public void init()
    {
        outputArea = new JTextArea();
        Container container = getContentPane();
        container.add( outputArea );

        int array1[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
        int array2[][] = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
        outputArea.setText( "Values in array1 by row are\n" );
        buildOutput( array1 );
    }
}
```

Fig. 7.15 Initializing multidimensional arrays (part 1 of 2).
Chapter 7 Arrays 345

The program declares two arrays in method `init`. The declaration of `array1` (line 20) provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3. The second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of `array2` (line 21) provides six initializers in three sublists. The sublist for the first row explicitly initializes the first row to have two elements with values 1 and 2, respectively. The sublist for the second row initializes the second row to have one element with value 3. The sublist for the third row initializes the third row to the values 4, 5 and 6.

Line 24 of method `init` calls method `buildOutput` (defined at lines 31–44) to append each array’s elements to `outputArea` (a `JTextArea`). Method `buildOutput` specifies the array parameter as `int array[][]` to indicate that the method receives a double-subscripted array as an argument. Note the use of a nested `for` structure (lines 34–43) to output the rows of a double-subscripted array. In the outer `for` structure, the expression `array.length` determines the number of rows in the array. In the inner `for` structure, the expression `array[row].length` determines the number of columns in the current row of the array. This condition enables the loop to determine the exact number of columns in each row.

![Applet Viewer: InitArray class](image)

Fig. 7.15 Initializing multidimensional arrays (part 2 of 2).

The program declares two arrays in method `init`. The declaration of `array1` (line 20) provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3. The second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of `array2` (line 21) provides six initializers in three sublists. The sublist for the first row explicitly initializes the first row to have two elements with values 1 and 2, respectively. The sublist for the second row initializes the second row to have one element with value 3. The sublist for the third row initializes the third row to the values 4, 5 and 6.

Line 24 of method `init` calls method `buildOutput` (defined at lines 31–44) to append each array’s elements to `outputArea` (a `JTextArea`). Method `buildOutput` specifies the array parameter as `int array[][]` to indicate that the method receives a double-subscripted array as an argument. Note the use of a nested `for` structure (lines 34–43) to output the rows of a double-subscripted array. In the outer `for` structure, the expression `array.length` determines the number of rows in the array. In the inner `for` structure, the expression `array[row].length` determines the number of columns in the current row of the array. This condition enables the loop to determine the exact number of columns in each row.
Many common array manipulations use for repetition structures. For example, the following for structure sets all the elements in the third row of array \( a \) in Fig. 7.14 to zero:

```c
for ( int column = 0; column < a[ 2 ].length; column++ )
    a[ 2 ][ column ] = 0;
```

We specified the third row; therefore, we know that the first subscript is always 2 (0 is the first row and 1 is the second row). The for loop varies only the second subscript (i.e., the column subscript). The preceding for structure is equivalent to the assignment statements

\[
\begin{align*}
a[ 2 ][ 0 ] &= 0; \\
a[ 2 ][ 1 ] &= 0; \\
a[ 2 ][ 2 ] &= 0; \\
a[ 2 ][ 3 ] &= 0;
\end{align*}
\]

The following nested for structure totals the values of all the elements in array \( a \).

```c
int total = 0;
for ( int row = 0; row < a.length; row++ )
    for ( int column = 0; column < a[ row ].length; column++ )
        total += a[ row ][ column ];
```

The for structure totals the elements of the array one row at a time. The outer for structure begins by setting the row subscript to 0 so that the elements of the first row may be totaled by the inner for structure. The outer for structure then increments row to 1 so that the second row can be totaled. Then, the outer for structure increments row to 2 so that the third row can be totaled. The result can be displayed when the nested for structure terminates.

The applet of Fig. 7.16 performs several other common array manipulations on 3-by-4 array \( \text{grades} \). Each row of the array represents a student, and each column represents a grade on one of the four exams the students took during the semester. Four methods perform the array manipulations. Method minimum (lines 52–69) determines the lowest grade of any student for the semester. Method maximum (lines 72–89) determines the highest grade of any student for the semester. Method average (lines 93–103) determines a particular student’s semester average. Method buildString (lines 106–121) appends the double-subscripted array to String\( \text{output} \) in a tabular format.

Methods minimum, maximum and buildString each use array \( \text{grades} \) and the variables students (number of rows in the array) and exams (number of columns in the array). Each method loops through array \( \text{grades} \) by using nested for structures—for example, the nested for structure from the definition of method minimum (lines 58–66). The outer for structure sets row (the row subscript) to 0 so that the elements of the first row can be compared to variable lowGrade in the body of the inner for structure. The inner for structure loops through the four grades of a particular row and compares each grade to lowGrade. If a grade is less than lowGrade, lowGrade is set to that grade. The outer for structure then increments the row subscript by 1. The elements of the second row are compared to variable lowGrade. The outer for structure then increments the row subscript to 2. The elements of the third row are compared to variable lowGrade. When
execution of the nested structure is complete, `lowGrade` contains the smallest grade in the double-subscripted array. Method `maximum` works similarly to method `minimum`.

```java
// Fig. 7.16: DoubleArray.java
// Double-subscripted array example

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

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

public class DoubleArray extends JApplet {

  int grades[][], students, exams;
  String output;
  JTextArea outputArea;

  // initialize instance variables
  public void init() {
    students = grades.length;     // number of students
    exams = grades[0].length;     // number of exams

    // create JTextArea and attach to applet
    outputArea = new JTextArea();
    Container container = getContentPane();
    container.add(outputArea);

    // build output string
    output = "The array is:
    buildString();

    // call methods minimum and maximum
    output += "\n\nLowest grade: " + minimum() + 
             "\nHighest grade: " + maximum() + "\n";

    // call method average to calculate each student's average
    for ( int counter = 0; counter < students; counter++ )
      output += "\nAverage for student " + counter + " is " + 
                average( grades[ counter ] );

    // change outputArea's display font
    outputArea.setFont( 
                         new Font("Courier", Font.PLAIN, 12 ) );

    // place output string in outputArea
    outputArea.setText( output );
  }
}
```

Fig. 7.16 Example of using double-subscripted arrays (part 1 of 3).
50 // find minimum grade
51 public int minimum()
52 {
53     // assume first element of grades array is smallest
54     int lowGrade = grades[ 0 ][ 0 ];
55
56     // loop through rows of grades array
57     for ( int row = 0; row < students; row++ )
58         // loop through columns of current row
59         for ( int column = 0; column < exams; column++ )
60             // Test if current grade is less than lowGrade.
61             // If so, assign current grade to lowGrade.
62             if ( grades[ row ][ column ] < lowGrade )
63                 lowGrade = grades[ row ][ column ];
64
65     return lowGrade; // return lowest grade
66 }
67
68 // find maximum grade
69 public int maximum()
70 {
71     // assume first element of grades array is largest
72     int highGrade = grades[ 0 ][ 0 ];
73
74     // loop through rows of grades array
75     for ( int row = 0; row < students; row++ )
76         // loop through columns of current row
77         for ( int column = 0; column < exams; column++ )
78             // Test if current grade is greater than highGrade.
79             // If so, assign current grade to highGrade.
80             if ( grades[ row ][ column ] > highGrade )
81                 highGrade = grades[ row ][ column ];
82
83     return highGrade; // return highest grade
84 }
85
86 // determine average grade for particular
87 // student (or set of grades)
88 public double average( int setOfGrades[] )
89 {
90     int total = 0; // initialize total
91
92     // sum grades for one student
93     for ( int count = 0; count < setOfGrades.length; count++ )
94         total += setOfGrades[ count ];
95
96 Fig. 7.16 Example of using double-subscripted arrays (part 2 of 3).
Method `average` takes one argument—a single-subscripted array of test results for a particular student. When line 41 calls `average`, the argument is `grades[counter]`, which specifies that a particular row of the double-subscripted array `grades` should be passed to `average`. For example, the argument `grades[1]` represents the four values (a single-subscripted array of grades) stored in the second row of the double-subscripted array `grades`. Remember that, in Java, a double-subscripted array is an array with elements that are single-subscripted arrays. Method `average` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result as a `double` value.
7.10 (Optional Case Study) Thinking About Objects: Collaboration Among Objects

In this section, we concentrate on the collaborations (interactions) among objects. When two objects communicate with each other to accomplish a task, they are said to collaborate—objects do this by invoking one another’s operations. We say that objects send messages to other objects. We explain how messages are sent and received in Java in Section 10.22. A collaboration consists of an object of one class sending a particular message to an object of another class.

The message sent by the first object invokes an operation of the second object. In “Thinking About Objects” Section 6.17, we determined many of the operations of the classes in our system. In this section, we concentrate on the messages that invoke these operations. Figure 7.22 is the table of classes and verb phrases from Section 6.17. We have removed all the verb phrases in classes Elevator and Person that do not correspond to operations. The remaining phrases are our first estimate of the collaborations in our system. As we proceed through this and the remaining “Thinking About Objects” sections, we will discover additional collaborations.

We examine the list of verb phrases to determine the collaborations in our system. For example, class Elevator lists the phrase “resets elevator button.” To accomplish this task, an object of class Elevator sends the resetButton message to an object of class ElevatorButton (invoking the resetButton operation of ElevatorButton). Figure 7.23 lists all the collaborations that can be gleaned from our table of verb phrases. According to Fig. 7.22, the Elevator rings the Bell and opens (and closes) the ElevatorDoor, so we include a ringBell, openDoor and closeDoor message in Fig. 7.23. However, we must consider how the FloorDoor opens and close. According to the class diagram of Fig. 3.23, ElevatorShaft is the only class that associates with FloorDoor. The Elevator signals its arrival (we assume to the ElevatorShaft) by sending an elevatorArrived message. The ElevatorShaft responds to this message by resetting the appropriate FloorButton and turning on the appropriate Light—the ElevatorShaft sends resetButton and turnOnLight messages. At this point in the design, we may assume that the Elevator also signals its departure—that is, the ElevatorShaft sends an elevatorDeparted message to the ElevatorShaft, which then turns off the appropriate Light by sending it a turnOffLight message.

<table>
<thead>
<tr>
<th>Class</th>
<th>Verb phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>Resets elevator button, rings elevator bell, signals its arrival, signals its departure, opens its door, closes its door.</td>
</tr>
<tr>
<td>ElevatorShaft</td>
<td>Turns off light, turns on light, resets floor button.</td>
</tr>
</tbody>
</table>

Fig. 7.22 Verb phrases for each class exhibiting behaviors in simulation (part 1 of 2).

1. We refer to an object by using that object’s class name preceded by an article (“a,” “an” or “the”)—for example, the Elevator refers to the object of class Elevator. Our syntax avoids redundancy—i.e., we avoid repeating the phrase “an object of class...”
<table>
<thead>
<tr>
<th>Class</th>
<th>Verb phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Presses floor button, presses elevator button, rides elevator, enters elevator.</td>
</tr>
<tr>
<td>FloorButton</td>
<td>Summons (requests) elevator.</td>
</tr>
<tr>
<td>ElevatorButton</td>
<td>Signals elevator to move to opposite floor.</td>
</tr>
<tr>
<td>FloorDoor</td>
<td>Signals person to enter elevator (by opening).</td>
</tr>
<tr>
<td>ElevatorDoor</td>
<td>Signals person to exit elevator (by opening), opens floor door, closes floor door.</td>
</tr>
<tr>
<td>ElevatorModel</td>
<td>Creates person.</td>
</tr>
</tbody>
</table>

**Fig. 7.22** Verb phrases for each class exhibiting behaviors in simulation (part 2 of 2).

<table>
<thead>
<tr>
<th>An object of class</th>
<th>Sends the message...</th>
<th>To an object of class...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>resetButton</td>
<td>ElevatorButton</td>
</tr>
<tr>
<td></td>
<td>ringBell</td>
<td>Bell</td>
</tr>
<tr>
<td></td>
<td>elevatorArrived</td>
<td>ElevatorShaft</td>
</tr>
<tr>
<td></td>
<td>elevatorDeparted</td>
<td>ElevatorShaft</td>
</tr>
<tr>
<td></td>
<td>openDoor</td>
<td>ElevatorDoor</td>
</tr>
<tr>
<td></td>
<td>closeDoor</td>
<td>ElevatorDoor</td>
</tr>
<tr>
<td>ElevatorShaft</td>
<td>resetButton</td>
<td>FloorButton</td>
</tr>
<tr>
<td></td>
<td>turnOnLight</td>
<td>Light</td>
</tr>
<tr>
<td></td>
<td>turnOffLight</td>
<td>Light</td>
</tr>
<tr>
<td>Person</td>
<td>pressButton</td>
<td>FloorButton, ElevatorButton</td>
</tr>
<tr>
<td></td>
<td>enterElevator</td>
<td>Elevator</td>
</tr>
<tr>
<td></td>
<td>exitElevator</td>
<td>Elevator</td>
</tr>
<tr>
<td>FloorButton</td>
<td>requestElevator</td>
<td>Elevator</td>
</tr>
<tr>
<td>ElevatorButton</td>
<td>moveElevator</td>
<td>Elevator</td>
</tr>
<tr>
<td>FloorDoor</td>
<td>doorOpen</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>doorClose</td>
<td>Person</td>
</tr>
<tr>
<td>ElevatorDoor</td>
<td>doorOpen</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>doorClose</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>openDoor</td>
<td>FloorDoor</td>
</tr>
<tr>
<td></td>
<td>closeDoor</td>
<td>FloorDoor</td>
</tr>
</tbody>
</table>

**Fig. 7.23** Collaborations in the elevator system.

A **Person** may press either a **FloorButton** or the **ElevatorButton**. A **Person** object may enter and exit the **Elevator**. Therefore, a **Person** may send **pressButton**, **enterElevator** and **exitElevator** messages. A **FloorButton** requests, or summons, the **Elevator**, so a **FloorButton** may send a **requestElevator** message. The **ElevatorButton** signals the **Elevator** to begin moving to the other floor, so the **ElevatorButton** may send a **moveElevator** message.
Both a FloorDoor and the ElevatorDoor inform a Person that they have opened or closed, so both objects send doorOpened and doorClosed messages. Finally, the ElevatorModel creates a Person, but the ElevatorModel cannot send a personCreated message, because that Person does not yet exist. We discover, in Section 8.6, how to use a special method that creates, or instantiates, new objects. This method is referred to as an constructor—the ElevatorModel will create a Person by calling that Person’s constructor (which is similar to sending the personCreated message).

Lastly, the ElevatorDoor must send openDoor and closeDoor messages to a FloorDoor to guarantee that these doors open and close together.

Collaboration Diagrams
Now let us consider the objects that must interact so that people in our simulation can enter and exit the elevator when it arrives on a floor. The UML provides the collaboration diagram to model such interactions. Collaboration diagrams are a type of interaction diagram; they model the behavioral aspects of the system by providing information about how objects interact. Collaboration diagrams emphasize which objects participate in the interactions. The other type of interaction diagram is the sequence diagram, which we present in Chapter 15. Figure 7.24 shows a collaboration diagram that models a person who is pressing a floor button. An object in a collaboration diagram is represented as a rectangle that encloses the object’s name. We write object names in the collaboration diagram by using the convention we introduced in the object diagram of Fig. 3.24—objects are written in the form objectName : ClassName. In this example, we disregard the object name, because we care only about the object type. Collaborating objects are connected with solid lines, and messages are passed between objects along these lines in the direction shown by arrows. The name of the message, which appears next to the arrow, is the name of a method belonging to the receiving object—think of the name as a “service” that the receiving object (a “server”) provides for its sending objects (its “clients”).

![Collaboration diagram of a person pressing a floor button](image)

Fig. 7.24  Collaboration diagram of a person pressing a floor button

The arrow in Fig. 7.24 represents a message in the UML and a method—or synchronous call—in Java. This arrow indicates that the flow of control is from the sending object to the receiving object—the sending object may not send another message until the receiving object processes the message and returns control to the sending object. For example, in Fig. 7.24, a Person calls method pressButton of a FloorButton and

---

2. Note that most of the messages perform some specific action on the receiving object; for example, the Elevator resets the ElevatorButton. However, other messages inform receiving objects of events that have already happened; for example, the FloorDoor informs the Person that the FloorDoor has opened. In Section 10.22, we elaborate on the topic of events—for now, however, we proceed as if the two types of messages are indistinguishable.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
may not send another message to an object until `pressButton` has finished and returns control to that `Person`. If our program contains a `FloorButton` object called `firstFloorButton`, and we assume that the `Person` manages the flow of control, the Java code implementation in class `Person` that represents this collaboration diagram is

```
    firstFloorButton.pressButton();
```

Figure 7.25 shows a collaboration diagram that models the interactions among objects in the system while objects of class `Person` enter and exit the elevator. The collaboration begins when the `Elevator` arrives on a `Floor`. The number to the left of the message name indicates the order in which the message is passed. The sequence of messages in a collaboration diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message 1 and ends with message 4.2. The sequence of passing messages follows a nested structure—for example, message 1.1 is the first message nested in message 1, and message 3.2 is the second message nested in message 3. Message 3.2.1 would be the first message nested in message 3.2. A message may be passed only when all nested messages from the previous message have been passed. For example, in Fig. 7.25, the `Elevator` passes message 4 after message 3, message 3.1, message 3.1.1, message 3.1.1.1, message 3.2 and message 3.2.1 are passed.

---

**Fig. 7.25** Collaboration diagram for passengers exiting and entering the elevator.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
The Elevator sends the resetButton message (message 1) to the ElevatorButton to reset the ElevatorButton. The Elevator sends the ringBell message (message 2) to the Bell, then opens the ElevatorDoor by passing the openDoor message (message 3). The ElevatorDoor then opens the FloorDoor by sending an openDoor message (message 3.1 at the top of the diagram) to that FloorDoor. The FloorDoor informs the waitingPassenger that the FloorDoor has opened (message 3.1.1), and the waitingPassenger enters the Elevator (message 3.1.1.1). The ElevatorDoor then informs the ridingPassenger that the ElevatorDoor has opened (message 3.2), so that the ridingPassenger may exit the Elevator (message 3.2.1). Lastly, the Elevator informs the ElevatorShaft of the arrival (message 4), so that the ElevatorShaft can reset the FloorButton (message 4.1) and turn on the Light (message 4.2).

Unfortunately, this design creates a problem. According to the diagram, the waitingPassenger enters the Elevator (message 3.1.1.1) before the ridingPassenger (message 3.2.1) exits. In “Thinking About Objects” Section 15.12, we apply multithreading, synchronization and active classes to our collaboration diagram, to force the waitingPassenger to wait for the ridingPassenger to exit the Elevator. Before we correct this problem, we modify this diagram to indicate more accurately the message passing in Section 10.2.2 when we discuss event handling.

**SUMMARY**

- Java stores lists of values in arrays. An array is a contiguous group of related memory locations. These locations are related by the fact that they all have the same name and the same type. To refer to a particular location or element within the array, we specify the name of the array and the subscript (or index or position number) of the element.
- Each array a length member that is set to the number of elements in the array at the time the program creates the array object.
- A subscript may be an integer or an integer expression. If a program uses an expression as a subscript, the program evaluates the expression to determine the particular element of the array.
- Java arrays always begin with element 0; thus, it is important to note the difference when referring to the “seventh element of the array” as opposed to “array element seven.” The seventh element has a subscript of 6, while array element seven has a subscript of 7 (actually the eighth element of the array).
- Arrays occupy space in memory and are considered to be objects. Operator new must be used to reserve space for an array. For example, the following creates an array of 100 int values:

  ```java
  int b[] = new int[100];
  ```

- When declaring an array, the type of the array and the square brackets can be combined at the beginning of the declaration to indicate that all identifiers in the declaration represent arrays, as in

  ```java
  double[] array1, array2;
  ```

- The elements of an array can be initialized with initializer lists in a declaration and by input.
- Java prevents referencing elements beyond the bounds of an array. If this occurs during program execution, an ArrayIndexOutOfBoundsException occurs.
- Constant variables must be initialized with a constant expression before they are used and cannot be modified thereafter.
• To pass an array to a method, pass the name of the array. To pass a single element of an array to a method, simply pass the name of the array followed by the subscript of the particular element.

• Arrays are passed to methods as references—therefore, the called methods can modify the element values in the caller’s original arrays. Single elements of primitive-data-type arrays are passed to methods by value.

• To receive an array argument, the method must specify an array parameter in the parameter list.

• An array can be sorted by using the bubble-sort technique. Several passes of the array are made. On each pass, successive pairs of elements are compared. If a pair is in order (or the values are identical), it is left as is. If a pair is out of order, the values are swapped. For small arrays, the bubble sort is acceptable, but for larger arrays it is inefficient compared to more sophisticated sorting algorithms.

• The linear search compares each element of the array with the search key. If the array is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program will have to compare the search key with half the elements of the array. Linear search works well for small arrays and is acceptable even for large unsorted arrays.

• For sorted arrays, the binary search eliminates from consideration one half of the elements in the array after each comparison. The algorithm locates the middle element of the array and compares it with the search key. If they are equal, the search key is found and the array subscript of that element is returned. Otherwise, the problem is reduced to searching one half of the array that is still under consideration. In a worst-case scenario, searching a sorted array of 1024 elements will take only 10 comparisons using a binary search.

• Most GUI components have method `setFont` to change the font of the text on the GUI component. The method requires a `Font` (package `java.awt`) object as its argument.

• A `Font` object is initialized with three arguments—a `String` representing the name of the font, an `int` representing the style of the font and an `int` representing the point size of the font. The style can be `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` or `Font.BOLD + Font.ITALIC`. The point size represents the size of the font—there are 72 points to an inch. The actual screen size may vary based on the size of the screen and the screen resolution.

• Arrays may be used to represent tables of values consisting of information arranged in rows and columns. To identify a particular element of a table, two subscripts are specified: The first identifies the row in which the element is contained, and the second identifies the column in which the element is contained. Tables or arrays that require two subscripts to identify a particular element are called double-subscripted arrays.

• A double-subscripted array can be initialized with an initializer list of the form

\[
arrayType\ \ arrayName[][]\ =\ \{\ \{\ row1\ sublist\ \},\ \{\ row2\ sublist\ \},\ \ldots\ \};
\]

• To dynamically create an array with a fixed number of rows and columns, use

\[
arrayType\ \ arrayName[][]\ =\ new\ arrayType[\ numRows][\ numColumns];
\]

• To pass one row of a double-subscripted array to a method that receives a single-subscripted array, simply pass the name of the array followed by only the row subscript.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a[i]</code></td>
<td>bubble sort</td>
</tr>
<tr>
<td><code>a[i][j]</code></td>
<td>column subscript</td>
</tr>
<tr>
<td>array</td>
<td>constant variable</td>
</tr>
<tr>
<td>array initializer list</td>
<td>declare an array</td>
</tr>
<tr>
<td>binary search of an array</td>
<td>double-subscripted array</td>
</tr>
<tr>
<td>bounds checking</td>
<td>element of an array</td>
</tr>
</tbody>
</table>
SELF-REVIEW EXERCISES

7.1 Fill in the blank(s) in each of the following statements:
   a) Lists and tables of values can be stored in ________.
   b) The elements of an array are related by the fact that they have the same ________ and ________.
   c) The number used to refer to a particular element of an array is called its ________.
   d) The process of placing the elements of an array in order is called ________ the array.
   e) Determining if an array contains a certain key value is called ________ the array.
   f) An array that uses two subscripts is referred to as a ________ array.

7.2 State whether each of the following is true or false. If false, explain why.
   a) An array can store many different types of values.
   b) An array subscript should normally be of data type float.
   c) An individual array element that is passed to a method and modified in that method will contain the modified value when the called method completes execution.

7.3 Answer the following questions regarding an array called fractions:
   a) Define a constant variable ARRAY_SIZE initialized to 10.
   b) Declare an array with ARRAY_SIZE elements of type float and initialize the elements to 0.
   c) Name the fourth element of the array.
   d) Refer to array element 4.
   e) Assign the value 1.667 to array element 9.
   f) Assign the value 3.333 to the seventh element of the array.
   g) Sum all the elements of the array using a for repetition structure. Define the integer variable x as a control variable for the loop.

7.4 Answer the following questions regarding an array called table:
   a) Declare and create the array as an integer array and with 3 rows and 3 columns. Assume the constant variable ARRAY_SIZE has been defined to be 3.
   b) How many elements does the array contain?
   c) Use a for repetition structure to initialize each element of the array to the sum of its subscripts. Assume the integer variables x and y are declared as control variables.
Chapter 7  Arrays  357

7.5  Find the error in each of the following program segments and correct the error:

a)  \texttt{final int ARRAY\_SIZE = 5;}
    \texttt{ARRAY\_SIZE = 10;}
b)  Assume \texttt{int b[]} = \texttt{new int[ 10 ];}
    \texttt{for ( int i = 0; i <= b.length; i++ )}
    \texttt{b[ i ] = 1;}
c)  Assume \texttt{int a[][]} = \{ \{ 1, 2 \}, \{ 3, 4 \} \};
    \texttt{a[ 1, 1 ] = 5;}

\textbf{ANSWERS TO SELF-REVIEW EXERCISES}

7.1  a) arrays.  b) name, type.  c) subscript (or index or position number).  d) sorting.  e) searching.  f) double-subscripted (or two-dimensional).

7.2  a) False. An array can store only values of the same type.
    b) False. An array subscript must be an integer or an integer expression.
    c) False for individual primitive-data-type elements of an array because they are passed by value. If a reference to an array is passed, then modifications to the array elements are reflected in the original. Also, an individual element of a nonprimitive type is passed by reference, and changes to the object will be reflected in the original array element.

7.3  a) \texttt{final int ARRAY\_SIZE = 10;}
b) \texttt{float fractions[] = new float[ ARRAY\_SIZE ];}
c) \texttt{fractions[ 3 ]}
d) \texttt{fractions[ 4 ]}
e) \texttt{fractions[ 9 ] = 1.667;}
f) \texttt{fractions[ 6 ] = 3.333;}
g) \texttt{float total = 0;}
    \texttt{for ( int x = 0; x < fractions.length; x++ )}
    \texttt{total += fractions[ x ];}

7.4  a) \texttt{int table[][] = new int[ ARRAY\_SIZE ][ ARRAY\_SIZE ];}
b) Nine.
c) \texttt{for ( int x = 0; x < table.length; x++ )}
    \texttt{for ( int y = 0; y < table[ x ].length; y++ )}
    \texttt{table[ x ][ y ] = x + y;}

7.5  The solutions to Exercise 7.5 are as follows:

a)  Error: Assigning a value to a constant variable after it has been initialized.
    Correction: Assign the correct value to the constant variable in a \texttt{final int ARRAY\_SIZE} declaration or create another variable.
b)  Error: Referencing an array element outside the bounds of the array (\texttt{b[10]}).  
    Correction: Change the \texttt{<=} operator to \texttt{<}.
c)  Error: Array subcripting performed incorrectly.
    Correction: Change the statement to \texttt{a[ 1 ][ 1 ] = 5;}.

\textbf{EXERCISES}

7.6  Fill in the blanks in each of the following:

a)  Java stores lists of values in ________.
b)  The elements of an array are related by the fact that they ________.
c)  When referring to an array element, the position number contained within brackets is called a ________.
d) The names of the four elements of array \( p \) are ________, ________, ________ and ________.

e) Naming an array, stating its type and specifying the number of dimensions in the array is called ________ the array.

f) The process of placing the elements of an array into either ascending or descending order is called ________.

g) In a double-subscripted array, the first subscript identifies the ________ of an element and the second subscript identifies the ________ of an element.

h) An \( m \)-by-\( n \) array contains ________ rows, ________ columns and ________ elements.

i) The name of the element in row 3 and column 5 of array \( d \) is ________.

7.7 State whether each of the following is true or false. If false, explain why.

a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.

b) An array declaration reserves space for the array.

c) To indicate that 100 locations should be reserved for integer array \( p \), the programmer writes the declaration

\[
p[100];
\]

d) A Java program that initializes the elements of a 15-element array to zero must contain at least one \texttt{for} statement.

e) A Java program that totals the elements of a double-subscripted array must contain nested \texttt{for} statements.

7.8 Write Java statements to accomplish each of the following:

a) Display the value of the seventh element of character array \( f \).

b) Initialize each of the five elements of single-subscripted integer array \( g \) to 8.

c) Total the elements of floating-point array \( c \) of 100 elements.

d) Copy 11-element array \( a \) into the first portion of array \( b \), containing 34 elements.

e) Determine and print the smallest and largest values contained in 99-element floating-point array \( w \).

7.9 Consider a 2-by-3 integer array \( t \).

a) Write a statement that declares and creates \( t \).

b) How many rows does \( t \) have?

c) How many columns does \( t \) have?

d) How many elements does \( t \) have?

e) Write the names of all the elements in the second row of \( t \).

f) Write the names of all the elements in the third column of \( t \).

g) Write a single statement that sets the element of \( t \) in row 1 and column 2 to zero.

h) Write a series of statements that initializes each element of \( t \) to zero. Do not use a repetition structure.

i) Write a nested \texttt{for} structure that initializes each element of \( t \) to zero.

j) Write a nested \texttt{for} structure that inputs the values for the elements of \( t \) from the keyboard.

k) Write a series of statements that determines and prints the smallest value in array \( t \).

l) Write a statement that displays the elements of the first row of \( t \).

m) Write a statement that totals the elements of the fourth column of \( t \).

n) Write a series of statements that prints the array \( t \) in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

7.10 Use a single-subscripted array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for
that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9% of $5000 or a total of $650. Write an applet (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson’s salary is truncated to an integer amount):

a) $200-$299  
b) $300-$399  
c) $400-$499  
d) $500-$599  
e) $600-$699  
f) $700-$799  
g) $800-$899  
h) $900-$999  
i) $1000 and over

The applet should use the GUI techniques introduced in Chapter 6. Display the results in a JTextArea. Use JTextArea method setText to update the results after each value input by the user.

7.11 The bubble sort presented in Fig. 7.11 is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort:

a) After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are “in place”; and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass and so on.

b) The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass if any swaps have been made. If none have been made, the data must already be in the proper order, so the program should terminate. If swaps have been made, at least one more pass is needed.

7.12 Write statements that perform the following single-subscripted array operations:

a) Set the 10 elements of integer array counts to zeros.

b) Add 1 to each of the 15 elements of integer array bonus.

c) Print the five values of integer array bestScores in column format.

7.13 Use a single-subscripted array to solve the following problem: Write an applet that inputs 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, display it only if it is not a duplicate of a number already read. Provide for the “worst case” in which all 20 numbers are different. Use the smallest possible array to solve this problem. The applet should use the GUI techniques introduced in Chapter 6. Display the results in a JTextArea. Use JTextArea method setText to update the results after each value input by the user.

7.14 Label the elements of 3-by-5 double-subscripted array sales to indicate the order in which they are set to zero by the following program segment:

```java
for ( int row = 0; row < sales.length; row++ )
    for ( int col = 0; col < sales[row].length; col++ )
        sales[row][col] = 0;
```

7.15 Write an applet to simulate the rolling of two dice. The program should use Math.random to roll the first die and should use Math.random again to roll the second die. The sum of the two values should then be calculated. [Note: Each die can show an integer value from 1 to 6, so the sum of the values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums. Figure 7.26 shows the 36 possible combinations of the two dice. Your program should
roll the dice 36,000 times. Use a single-subscripted array to tally the numbers of times each possible sum appears. Display the results in a JTextArea in tabular format. Also, determine whether the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one sixth of all the rolls should be 7). The applet should use the GUI techniques introduced in Chapter 6. Provide a JButton to allow the user of the applet to roll the dice another 36,000 times. The applet should reset the elements of the single-subscripted array to 0 before rolling the dice again.

### 7.16 What does the program of Fig. 7.27 do?

```java
// Exercise 7.16: WhatDoesThisDo.java
// Java core packages
import java.awt.*;

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

public class WhatDoesThisDo extends JApplet {
    int result;

    public void init() {
        int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        result = whatIsThis( array, array.length );

        Container container = getContentPane();
        JTextArea output = new JTextArea();
        output.setText( "Result is: " + result );
        container.add( output );
    }

    public int whatIsThis( int array2[], int size ) {
        if ( size == 1 )
            return array2[ 0 ];
    }
}
```

**Fig. 7.26** The 36 possible outcomes of rolling two dice.

**Fig. 7.27** Determine what this program does.
7.17 Write a program that runs 1000 games of craps (Fig. 6.9) and answers the following questions:

a) How many games are won on the first roll, second roll, …, twentieth roll and after the twentieth roll?

b) How many games are lost on the first roll, second roll, …, twentieth roll and after the twentieth roll?

c) What are the chances of winning at craps? \[\text{Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?}\]

d) What is the average length of a game of craps?

e) Do the chances of winning improve with the length of the game?

7.18 (Airline Reservations System) A small airline has just purchased a computer for its new automated reservations system. You have been asked to program the new system. You are to write an applet to assign seats on each flight of the airline’s only plane (capacity: 10 seats).

Your program should display the following alternatives:

Please type 1 for "smoking"
Please type 2 for "nonsmoking"

If the person types 1, your program should assign a seat in the smoking section (seats 1-5). If the person types 2, your program should assign a seat in the nonsmoking section (seats 6-10). Your program should then print a boarding pass indicating the person’s seat number and whether it is in the smoking or nonsmoking section of the plane.

Use a single-subscripted array of primitive type boolean to represent the seating chart of the plane. Initialize all the elements of the array to false to indicate that all seats are empty. As each seat is assigned, set the corresponding elements of the array to true to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the smoking section is full, your program should ask the person if it is acceptable to be placed in the nonsmoking section (and vice versa). If yes, make the appropriate seat assignment. If no, print the message "Next flight leaves in 3 hours."

7.19 What does the program of Fig. 7.28 do?

```java
// Exercise 7.19: WhatDoesThisDo2.java
// Java core packages
import java.awt.*;

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

// Exercise 7.19: WhatDoesThisDo2.java
```
7.20 Use a double-subscripted array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains the following:

a) The salesperson number
b) The product number
c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write an applet that will read all this information for last month’s sales and summarize the total sales by salesperson by product. All totals should be stored in the double-subscripted array `sales`. After processing all the information for last month, display the results in tabular format with each of the columns representing a particular salesperson and each of the rows representing a particular product. Cross total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns. Display the results in a `JTextArea`.

7.21 (Turtle Graphics) The Logo language, which is popular among young computer users, made the concept of turtle graphics famous. Imagine a mechanical turtle that walks around the room under the control of a Java program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about freely without writing anything. In this problem you will simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 20-by-20 array `floor` that is initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position 0,0 of the floor with its pen up. The set of turtle commands your program must process are shown in Fig. 7.29.
Suppose that the turtle is somewhere near the center of the floor. The following “program” would draw and print a 12-by-12 square, leaving the pen in the up position:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array `floor` to 1s. When the 6 command (print) is given, wherever there is a 1 in the array, display an asterisk or some other character you choose. Wherever there is a zero, display a blank. Write a Java applet to implement the turtle graphics capabilities discussed here. The applet should display the turtle graphics in a `JTextArea`, using Monospaced font. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

### 7.22 (Knight’s Tour)

One of the more interesting puzzlers for chess buffs is the Knight’s Tour problem, originally proposed by the mathematician Euler. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth here.

The knight makes L-shaped moves (over two in one direction and then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 7.30.

a) Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight’s Tour by hand. Put a 1 in the first square you move to, a 2 in the second square, a 3 in the third, etc. Before starting the tour, estimate how far you think you will get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?
b) Now let us develop an applet that will move the knight around a chessboard. The board is represented by an 8-by-8 double-subscripted array \texttt{board}. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their horizontal and vertical components. For example, a move of type 0 as shown in Fig. 7.30 consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two single-subscripted arrays, \texttt{horizontal} and \texttt{vertical}, as follows:

\begin{verbatim}
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
\end{verbatim}

\textbf{Fig. 7.30} The eight possible moves of the knight.
Let the variables `currentRow` and `currentColumn` indicate the row and column of the knight’s current position. To make a move of type `moveNumber`, where `moveNumber` is between 0 and 7, your program uses the statements

```plaintext
    currentRow += vertical[ moveNumber ];
    currentColumn += horizontal[ moveNumber ];
```

Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Test each potential move to see if the knight already visited that square. Test every potential move to ensure that the knight does not land off the chessboard. Write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?

c) After attempting to write and run a Knight’s Tour program, you have probably developed some valuable insights. We will use these to develop a heuristic (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome or inaccessible squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to so when the board gets congested near the end of the tour there will be a greater chance of success.

We could develop an “accessibility heuristic” by classifying each of the squares according to how accessible they are, then always moving the knight (using the knight’s L-shaped moves) to the most inaccessible square. We label a double-subscripted array `accessibility` with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each center square is rated as 8, each corner square is rated as 2 and the other squares have accessibility numbers of 3, 4 or 6 as follows:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Write a version of the Knight’s Tour using the accessibility heuristic. The knight should always move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your program should reduce the accessibility numbers as more squares become occupied. In this way, at any given time during the tour, each available square’s accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your program. Did you get a full tour? Modify the program to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

d) Write a version of the Knight’s Tour program that, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the “tied” squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.
7.23 (Knight’s Tour: Brute Force Approaches) In Exercise 7.22, we developed a solution to the Knight’s Tour problem. The approach used, called the “accessibility heuristic,” generates many solutions and executes efficiently.

As computers continue increasing in power, we will be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. Let us call this approach “brute force” problem solving.

a) Use random number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?

b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a single-subscripted array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?

c) Most likely, the preceding program gave you some “respectable” tours, but no full tours. Now “pull all the stops out” and simply let your program run until it produces a full tour. (Caution: This version of the program could run for hours on a powerful computer.) Once again, keep a table of the number of tours of each length and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?

d) Compare the brute force version of the Knight’s Tour with the accessibility-heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility-heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute force approach? Argue the pros and cons of brute force problem solving in general.

7.24 (Eight Queens) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is “attacking” any other, i.e., no two queens are in the same row, in the same column or along the same diagonal? Use the thinking developed in Exercise 7.22 to formulate a heuristic for solving the Eight Queens problem. Run your program. (Hint: It is possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are “eliminated” if a queen is placed in that square. Each of the corners would be assigned the value 22, as in Figure 7.31.) Once these “elimination numbers” are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

7.25 (Eight Queens: Brute Force Approaches) In this exercise, you will develop several brute force approaches to solving the Eight Queens problem introduced in Exercise 7.24.

a) Solve the Eight Queens exercise, using the random brute force technique developed in Exercise 7.23.

b) Use an exhaustive technique (i.e., try all possible combinations of eight queens on the chessboard).

c) Why do you suppose the exhaustive brute force approach may not be appropriate for solving the Knight’s Tour problem?

d) Compare and contrast the random brute force and exhaustive brute force approaches.

7.26 (Knight’s Tour: Closed Tour Test) In the Knight’s Tour, a full tour occurs when the knight makes 64 moves touching each square of the chessboard once and only once. A closed tour occurs when the 64th move is one move away from the square in which the knight started the tour. Modify the program you wrote in Exercise 7.22 to test for a closed tour if a full tour has occurred.

7.27 (The Sieve of Eratosthenes) A prime integer is any integer that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:
a) Create a primitive type boolean array with all elements initialized to true. Array elements with prime subscripts will remain true. All other array elements will eventually be set to false.

b) Starting with array subscript 2 (subscript 1 must be prime), determine whether a given element is true. If so, loop through the remainder of the array and set to false every element whose subscript is a multiple of the subscript for the element with value true. Then, continue the process with the next element with value true. For array subscript 2, all elements beyond element 2 in the array that have subscripts which are multiples of 2 will be set to false (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond element 3 in the array that have subscripts which are multiples of 3 will be set to false (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still true indicate that the subscript is a prime number. These subscripts can be displayed. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

7.28 (Bucket Sort) A bucket sort begins with a single-subscripted array of positive integers to be sorted and a double-subscripted array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to \( n - 1 \), where \( n \) is the number of values in the array to be sorted. Each row of the double-subscripted array is referred to as a bucket. Write an applet containing a method called bucketSort that takes an integer array as an argument and performs as follows:

a) Place each value of the single-subscripted array into a row of the bucket array based on the value’s ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This is called a “distribution pass.”

b) Loop through the bucket array row by row and copy the values back to the original array. This is called a “gathering pass.” The new order of the preceding values in the single-subscripted array is 100, 3 and 97.

c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the single-subscripted array is 100, 3 and 97. On the third pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After the last gathering pass, the original array is now in sorted order.

Note that the double-subscripted array of buckets is ten times the size of the integer array being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory. The bubble sort requires space for only one additional element of data. This is an example of the space-time trade-off: The bucket sort uses more memory than the bubble sort, but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second double-subscripted bucket array and repeatedly swap the data between the two bucket arrays.
**RECURSION EXERCISES**

7.29 *(Selection Sort)* A selection sort searches an array looking for the smallest element in the array, then swaps that element with the first element of the array. The process is repeated for the subarray beginning with the second element. Each pass of the array places one element in its proper location. For an array of $n$ elements, $n - 1$ passes must be made, and for each subarray, $n - 1$ comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write recursive method `selectionSort` to perform this algorithm.

7.30 *(Palindromes)* A palindrome is a string that is spelled the same way forward and backward. Some examples of palindromes are “radar,” “able was i ere i saw elba” and (if blanks are ignored) “a man a plan a canal panama.” Write a recursive method `testPalindrome` that returns `boolean` value `true` if the string stored in the array is a palindrome and `false` otherwise. The method should ignore spaces and punctuation in the string. [Hint: Use `String` method `toCharArray`, which takes no arguments, to get a `char` array containing the characters in the `String`. Then, pass the array to method `testPalindrome`.

7.31 *(Linear Search)* Modify Figure 7.12 to use recursive method `linearSearch` to perform a linear search of the array. The method should receive an integer array, the array size and the search key as arguments. If the search key is found, return the array subscript; otherwise, return –1.

7.32 *(Binary Search)* Modify the program of Figure 7.13 to use a recursive method `binarySearch` to perform the binary search of the array. The method should receive an integer array and the starting subscript and ending subscript as arguments. If the search key is found, return the array subscript; otherwise, return –1.

7.33 *(Eight Queens)* Modify the Eight Queens program you created in Exercise 7.24 to solve the problem recursively.

7.34 *(Print an array)* Write a recursive method `printArray` that takes an array of `int` values and the size of the array as arguments and returns nothing. The method should stop processing and return when it receives an array of size 0.

7.35 *(Print a string backward)* Write a recursive method `stringReverse` that takes a character array containing a string as an argument, prints the string backward and returns nothing.

7.36 *(Find the minimum value in an array)* Write a recursive method `recursiveMinimum` that takes an integer array and the array size as arguments and returns the smallest element of the array. The method should stop processing and return when it receives an array of one element.

7.37 *(QuickSort)* In the examples and exercises of this chapter, we discussed the sorting techniques bubble sort, bucket sort and selection sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

a) **Partitioning Step:** Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.

b) **Recursive Step:** Perform step 1 on each unsorted subarray.

Each time step 1 is performed on a subarray, another element is placed in its final location of the sorted array and two unsorted subarrays are created. When a subarray consists of one element, it must be sorted therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

```
37  2  6  4  89  8  10  12  68  45
```
a) Starting from the rightmost element of the array, compare each element to 37 until an element less than 37 is found, then swap 37 and that element. The first element less than 37 is 12, so 37 and 12 are swapped. The new array is

12 2 6 4 89 8 10 37 68 45

Element 12 is in italic to indicate that it was just swapped with 37.

b) Starting from the left of the array, but beginning with the element after 12, compare each element to 37 until an element greater than 37 is found, then swap 37 and that element. The first element greater than 37 is 89, so 37 and 89 are swapped. The new array is

12 2 6 4 37 8 10 89 68 45

c) Starting from the right, but beginning with the element before 89, compare each element to 37 until an element less than 37 is found, then swap 37 and that element. The first element less than 37 is 10, so 37 and 10 are swapped. The new array is

12 2 6 4 10 8 37 89 68 45

d) Starting from the left, but beginning with the element after 10, compare each element to 37 until an element greater than 37 is found, then swap 37 and that element. There are no more elements greater than 37, so when we compare 37 to itself we know that 37 has been placed in its final location of the sorted array.

Once the partition has been applied on the previous array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive method `quickSort` to sort a single-subscripted integer array. The method should receive as arguments an integer array, a starting subscript and an ending subscript. Method `partition` should be called by `quickSort` to perform the partitioning step.

7.38 (Maze Traversal) The following grid of #s and dots (.) is a double-subscripted array representation of a maze.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

In the preceding double-subscripted array, the #s represent the walls of the maze, and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand
from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the
one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

Write recursive method \texttt{mazeTraverse} to walk through the maze. The method should
receive as arguments a 12-by-12 character array representing the maze and the starting location of
the maze. As \texttt{mazeTraverse} attempts to locate the exit from the maze, it should place the character \texttt{X} in each square in the path. The method should display the maze after each move so the user can
watch as the maze is solved.

7.39 (Generating Mazes Randomly) Write a method \texttt{mazeGenerator} that takes as an argument
a double-subscripted 12-by-12 character array and randomly produces a maze. The method should
also provide the starting and ending locations of the maze. Try your method \texttt{mazeTraverse} from
Exercise 7.38, using several randomly generated mazes.

7.40 (Mazes of Any Size) Generalize methods \texttt{mazeTraverse} and \texttt{mazeGenerator} of
Exercise 7.38 and Exercise 7.39 to process mazes of any width and height.

7.41 (Simulation: The Tortoise and the Hare) In this problem, you will recreate one of the truly
great moments in history, namely the classic race of the tortoise and the hare. You will use random
number generation to develop a simulation of this memorable event.

Our contenders begin the race at “square 1” of 70 squares. Each square represents a possible
position along the race course. The finish line is at square 70. The first contender to reach or pass
square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side
of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your applet should
adjust the position of the animals according to the rules in Fig. 7.32

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start
each animal at position 1 (i.e., the “starting gate”). If an animal slips left before square 1, move
the animal back to square 1.

Generate the percentages in the preceding table by producing a random integer, \( i \), in the range \( 1 \leq i \leq 10 \). For the tortoise, perform a “fast plod” when \( 1 \leq i \leq 5 \), a “slip” when \( 6 \leq i \leq 7 \) or a “slow
plod” when \( 8 \leq i \leq 10 \). Use a similar technique to move the hare.

Begin the race by printing

\textbf{BANG ! ! ! ! ! !}

\textbf{AND THEY'RE OFF ! ! ! ! ! !}

Then, for each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing
the letter \texttt{T} in the position of the tortoise and the letter \texttt{H} in the position of the hare. Occasionally, the
contenders will land on the same square. In this case, the tortoise bites the hare, and your program
should print \texttt{OUCH!!!} beginning at that position. All print positions other than the \texttt{T}, the \texttt{H} or the
\texttt{OUCH!!!} (in case of a tie) should be blank.

After each line is printed, test for whether either animal has reached or passed square 70. If so,
print the winner and terminate the simulation. If the tortoise wins, print \texttt{TORTOISE WINS!!!}
\texttt{YAY!!!} If the hare wins, print \texttt{Hare wins. Yuch.} If both animals win on the same tick of the
clock, you may want to favor the turtle (the “underdog”) or you may want to print \texttt{It's a tie}. If
neither animal wins, perform the loop again to simulate the next tick of the clock. When you are
ready to run your program, assemble a group of fans to watch the race. You’ll be amazed at how
involved your audience gets!

Later in the book, we introduce a number of Java capabilities, such as graphics, images, animation,
sound and multithreading. As you study those features, you might enjoy enhancing your tortoise
and hare contest simulation.
SPECIAL SECTION: BUILDING YOUR OWN COMPUTER

In the next several problems, we take a temporary diversion away from the world of high-level language programming. We “peel open” a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (through the technique of software-based simulation) on which you can execute your machine-language programs!

7.42 (Machine-Language Programming) Let us create a computer we will call the Simpletron. As its name implies, it is a simple machine, but as we will soon see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands, that is, Simpletron Machine Language or SML for short.

The Simpletron contains an accumulator—a “special register” in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number such as +3364, −1293, +0007, −0001, etc. The Simpletron is equipped with a 100-word memory and these words are referenced by their location numbers 00, 01, …, 99.

Before running an SML program, we must load or place the program into memory. The first instruction (or statement) of every SML program is always placed in location 00. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron’s memory (and hence instructions are signed four-digit decimal numbers). We shall assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron’s memory may contain either an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the operation code specifying the operation to be performed. SML operation codes are summarized in Fig. 7.33.

The last two digits of an SML instruction are the operand—the address of the memory location containing the word to which the operation applies. Let’s consider several simple SML programs.

Fig. 7.32 Rules for adjusting the positions of the tortoise and the hare.

<table>
<thead>
<tr>
<th>Animal</th>
<th>Move type</th>
<th>Percentage of the time</th>
<th>Actual move</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tortoise</td>
<td>Fast plod</td>
<td>50%</td>
<td>3 squares to the right</td>
</tr>
<tr>
<td></td>
<td>Slip</td>
<td>20%</td>
<td>6 squares to the left</td>
</tr>
<tr>
<td></td>
<td>Slow plod</td>
<td>30%</td>
<td>1 square to the right</td>
</tr>
<tr>
<td>Hare</td>
<td>Sleep</td>
<td>20%</td>
<td>No move at all</td>
</tr>
<tr>
<td></td>
<td>Big hop</td>
<td>20%</td>
<td>9 squares to the right</td>
</tr>
<tr>
<td></td>
<td>Big slip</td>
<td>10%</td>
<td>12 squares to the left</td>
</tr>
<tr>
<td></td>
<td>Small hop</td>
<td>30%</td>
<td>1 square to the right</td>
</tr>
<tr>
<td></td>
<td>Small slip</td>
<td>20%</td>
<td>2 squares to the left</td>
</tr>
</tbody>
</table>
The first SML program (Fig. 7.34) reads two numbers from the keyboard and computes and prints their sum. The instruction +1007 reads the first number from the keyboard and places it into location 07 (which has been initialized to 0). Then, instruction +1008 reads the next number into location 08. The load instruction, +2007, puts the first number into the accumulator, and the add instruction, +3008, adds the second number to the number in the accumulator. All SML arithmetic instructions leave their results in the accumulator. The store instruction, +2109, places the result back into memory location 09 from which the write instruction, +1109, takes the number and prints it (as a signed four-digit decimal number). The halt instruction, +4300, terminates execution.
The second SML program (Fig. 7.35) reads two numbers from the keyboard and determines and prints the larger value. Note the use of the instruction +4107 as a conditional transfer of control, much the same as Java’s if statement.

Now write SML programs to accomplish each of the following tasks:

a) Use a sentinel-controlled loop to read 10 positive numbers. Compute and print their sum.

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

c) Read a series of numbers and determine and print the largest number. The first number read indicates how many numbers should be processed.
7.43 (A Computer Simulator) It may at first seem outrageous, but in this problem you are going to build your own computer. No, you will not be soldering components together. Rather, you will use the powerful technique of software-based simulation to create an object-oriented software model of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you will actually be able to run, test and debug the SML programs you wrote in Exercise 7.42. Your Simpletron will be an event-driven applet—you will click a button to execute each SML instruction and you will be able to see the instruction “in action.”

When you run your Simpletron simulator, it should begin by displaying:

```plaintext
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time into the input ***
*** text field. I will display the location ***
*** number and a question mark (?). You then ***
*** type the word for that location. Press the ***
*** Done button to stop entering your program. ***
```

The program should display an input JTextField in which the user will type each instruction one at a time and a Done button for the user to click when the complete SML program has been entered. Simulate the memory of the Simpletron with a single-subscripted array memory that has 100 elements. Now assume that the simulator is running and let us examine the dialog as we enter the program of Fig. 7.35 (Exercise 7.42):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
```

Your program should use a JTextField to display the memory location followed by a question mark. Each of the values to the right of a question mark is typed by the user into the input JTextField. When the Done button is clicked, the program should display:

```plaintext
*** Program loading completed ***
*** Program execution begins ***
```

The SML program has now been placed (or loaded) in array memory. The Simpletron should provide an “Execute next instruction” button the user can click to execute each instruction in your SML program. Execution begins with the instruction in location 00 and, like Java, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable accumulator to represent the accumulator register. Use the variable instructionCounter to keep track of the location in memory that contains the instruction being performed. Use the variable operationCode to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use the variable operand to indicate the memory location on which the current instruction operates. Thus, operand is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called instructionRegister. Then “pick off” the left two digits and place them in operation-
Code and “pick off” the right two digits and place them in operand. Each of the preceding registers should have a corresponding JTextField in which its current value can be displayed at all times. When Simpletron begins execution, the special registers are all initialized to 0.

Now, let us “walk through” execution of the first SML instruction, +1009 in memory location 00. This is called an instruction execution cycle.

The instructionCounter tells us the location of the next instruction to be performed. We fetch the contents of that location from memory by using the Java statement

```
instructionRegister = memory[ instructionCounter ];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a read (versus a write, a load, etc.). A switch differentiates among the twelve operations of SML.

In the switch structure, the behavior of various SML instructions is simulated as shown in Fig. 7.36. We discuss branch instructions shortly and leave the others to the reader.

When the SML program completes execution, the name and contents of each register as well as the complete contents of memory should be displayed. Such a printout is often called a computer dump (and, no, a computer dump is not a place where old computers go). To help you program your dump method, a sample dump format is shown in Fig. 7.37. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. The sample dump assumes the output will be sent to the display screen with a series of System.out.print and System.out.println method calls. However, we encourage you to experiment with a version that can be displayed on the applet using a JTextArea or an array of JTextField objects.

Let us proceed with the execution of our program’s first instruction, namely the +1009 in location 00. As we have indicated, the switch statement simulates this by prompting the user to enter a value into the input dialog, reading the value, converting the value to an integer and storing it in memory location memory[ operand ]. Since your Simpletron is event driven, it waits for the user to type a value into the input JTextField and press the Enter key. The value is then read into location 09.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Since the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

![Fig. 7.36 Behavior of several SML instructions in the Simpletron.](Image)
This completes the simulated execution of the first instruction. When the user clicks the Execute next instruction button, the entire process (i.e., the instruction execution cycle) begins again with the fetch of the next instruction to be executed.

Now let us consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the switch as

\[
\text{instructionCounter} = \text{operand};
\]

The conditional “branch if accumulator is zero” instruction is simulated as

\[
\text{if ( accumulator == 0 )}
\]
\[
\text{instructionCounter} = \text{operand};
\]

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 7.42. You may embellish SML with additional features and provide for these in your simulator.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron’s memory must be in the range \(-9999\) to \(+9999\). Your simulator should test that each number entered is in this range, and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, accumulator overflows (i.e., arithmetic operations resulting in values larger than \(+9999\) or smaller than \(-9999\)) and the like. Such serious errors are called fatal errors. When a fatal error is detected, your simulator should print an error message such as:

*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
and should print a full computer dump in the format we discussed previously. This will help the user locate the error in the program.

7.44  (*Modifications to the Simpletron Simulator*) In Exercise 7.43, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise, we propose several modifications and enhancements to the Simpletron Simulator. In Exercise 19.26 and Exercise 19.27, we propose building a compiler that converts programs written in a high-level programming language (a variation of Basic) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler:

   a) Extend the Simpletron Simulator’s memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.

   b) Allow the simulator to perform modulus calculations. This requires an additional Simpletron Machine Language instruction.

   c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.

   d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.

   e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.

   f) Modify the simulator to process floating-point values in addition to integer values.

   g) Modify the simulator to handle string input. [*Hint:* Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a “half-word.”]

   h) Modify the simulator to handle output of strings stored in the format of part g). [*Hint:* Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

7.45   The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms.

   a) Write a nonrecursive method `fibonacci(n)` that calculates the `n`th Fibonacci number. Incorporate this method into an applet that enables the user to enter the value of `n`.

   b) Determine the largest Fibonacci number that can be printed on your system.

   c) Modify the program of part a) to use `double` instead of `int` to calculate and return Fibonacci numbers and use this modified program to repeat part b).
Object-Based Programming

Objectives

• To understand encapsulation and data hiding.
• To understand the notions of data abstraction and abstract data types (ADTs).
• To create Java ADTs, namely, classes.
• To be able to create, use and destroy objects.
• To be able to control access to object instance variables and methods.
• To appreciate the value of object orientation.
• To understand the use of the \texttt{this} reference.
• To understand class variables and class methods.

\begin{quote}
\textit{My object all sublime}
I shall achieve in time.
W. S. Gilbert
\end{quote}

\begin{quote}
\textit{Is it a world to hide virtues in?}
William Shakespeare, Twelfth Night
\end{quote}

\begin{quote}
\textit{Your public servants serve you right.}
Adlai Stevenson
\end{quote}

\begin{quote}
\textit{But what, to serve our private ends,}
\textit{Forbids the cheating of our friends?}
Charles Churchill
\end{quote}

\begin{quote}
\textit{This above all: to thine own self be true.}
William Shakespeare, Hamlet
\end{quote}

\begin{quote}
\textit{Have no friends not equal to yourself.}
Confucius
\end{quote}
8.1 Introduction

Now we investigate object orientation in Java in greater depth. Why did we defer this until now? First, the objects we will build will be composed in part of structured program pieces, so we needed to establish a basis in structured programming with control structures. Second, we wanted to study methods in depth. Third, we wanted to familiarize the reader with arrays that are Java objects.

Through our discussions of object-oriented Java programs in Chapter 2 through Chapter 7, we introduced many basic concepts (i.e., “object think”) and terminology (i.e., “object speak”) of object-oriented programming in Java. We also discussed our program-development methodology: We analyzed many typical problems that required a program—either a Java applet or a Java application—to be built, determined what classes from the Java API were needed to implement the program, determined what instance variables were needed, determined what methods were needed and specified how an object of our class collaborated with objects of Java API classes to accomplish the overall goals of the program.
Let us briefly review some key concepts and terminology of object orientation. OOP encapsulates data (attributes) and methods (behaviors) into objects; the data and methods of an object are intimately tied together. Objects have the property of information hiding. This means that although objects might know how to communicate with one another across well-defined interfaces, objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. Surely it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

In C and other procedural programming languages, programming tends to be action-oriented. In Java, programming is object-oriented. In C, the unit of programming is the function (called methods in Java). In Java, the unit of programming is the class from which objects are eventually instantiated (i.e., created). Functions do not disappear in Java; rather, they are encapsulated as methods with the data they process within the “walls” of classes.

C programmers concentrate on writing functions. Groups of actions that perform some task are formed into functions, and functions are grouped to form programs. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform. The verbs in a system-requirements document help the C programmer determine the set of functions that will work together to implement the system.

Java programmers concentrate on creating their own user-defined types called classes. Classes are also referred to as programmer-defined types. Each class contains data as well as the set of methods that manipulate the data. The data components of a class are called instance variables (these are called data members in C++). Just as an instance of a built-in type such as int is called a variable, an instance of a user-defined type (i.e., a class) is called an object. The focus of attention in Java is on objects rather than methods. The nouns in a system-requirements document help the Java programmer determine an initial set of classes with which to begin the design process. These classes are then used to instantiate objects that will work together to implement the system.

This chapter explains how to create and use objects, a subject called object-based programming (OBP). Chapter 9 introduces inheritance and polymorphism—two key technologies that enable true object-oriented programming (OOP). Although inheritance is not discussed in detail until Chapter 9, inheritance is part of every Java class definition.

Performance Tip 8.1
When passing an object to a method in Java, only a reference to the object is passed, not a copy of a possibly large object (as would be the case in a pass by value).

Software Engineering Observation 8.1
It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate their code’s being modified. As we will see, classes facilitate program modifiability.

8.2 Implementing a Time Abstract Data Type with a Class
The next example consists of two classes—Time1 (Fig. 8.1) and TimeTest (Fig. 8.2). Class Time1 is defined in file Time1.java. Class TimeTest is defined in a separate file called TimeTest.java. It is important to note that these classes must be defined in
separate files, because they are both public classes. [Note: The output of this program appears in Fig. 8.2.]

```java
// Fig. 8.1: Time1.java
// Time1 class definition maintains the time in 24-hour format.

import java.text.DecimalFormat;

public class Time1 extends Object {
    private int hour;  // 0 - 23
    private int minute;  // 0 - 59
    private int second;  // 0 - 59

    // Time1 constructor initializes each instance variable to zero. Ensures that each Tim1 object starts in a consistent state.
    public Time1()
    {
        setTime( 0, 0, 0 );
    }

    // Set a new time value using universal time. Perform validity checks on the data. Set invalid values to zero.
    public void setTime( int h, int m, int s )
    {
        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
    }

    // convert to String in universal-time format
    public String toUniversalString()
    {
        DecimalFormat twoDigits = new DecimalFormat( "00" );
        return twoDigits.format( hour ) + ":" +
                   twoDigits.format( minute ) + ":" +
                   twoDigits.format( second );
    }

    // convert to String in standard-time format
    public String toString()
    {
        DecimalFormat twoDigits = new DecimalFormat( "00" );
        return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
                ":" + twoDigits.format( minute ) +
                ":" + twoDigits.format( second ) +
                ( hour < 12 ? " AM" : " PM" );
    }
}
```
Software Engineering Observation 8.2

Class definitions that begin with keyword **public** must be stored in a file that has exactly the same name as the class and ends with the **.java** file name extension.

Common Programming Error 8.1

Defining more than one **public** class in the same file is a syntax error.

Figure 8.1 contains a simple definition for class **Time1**. Our **Time1** class definition begins with line 6, which indicates that class **Time1** extends class **Object** (from package **java.lang**). Remember that you never create a class definition “from scratch.” In fact, when you create a class definition, you always use pieces of an existing class definition. Java uses **inheritance** to create new classes from existing class definitions. Keyword **extends** followed by class name **Object** indicates the class (in this case **Time1**) from which our new class inherits existing pieces. In this inheritance relationship, **Object** is called the **superclass** or **base class** and **Time1** is called the **subclass** or **derived class**. Using inheritance results in a new class definition that has the attributes (data) and behaviors (methods) of class **Object** as well as new features we add in our **Time1** class definition. Every class in Java is a subclass of **Object** (directly or indirectly). Therefore, every class inherits the 11 methods defined by class **Object**. One key **Object** method is **toString**, discussed later in this section. Other methods of class **Object** are discussed as they are needed throughout the text. For a complete list of class Object’s methods, see the online API documentation at

`java.sun.com/j2se/1.3/docs/api/index.html`

Software Engineering Observation 8.3

Every class defined in Java must extend another class. If a class does not explicitly use the keyword **extends** in its definition, the class implicitly **extends Object**.

The **body** of the class definition is delineated with left and right braces (**{** and **}** on lines 7 and 50. Class **Time1** contains three integer instance variables—**hour**, **minute** and **second**—that represent the time in universal-time format (24-hour clock format).

Keywords **public** and **private** are **member access modifiers**. Instance variables or methods declared with member access modifier **public** are accessible wherever the program has a reference to a **Time1** object. Instance variables or methods declared with member access modifier **private** are accessible only to methods of the class in which they are defined. Every instance variable or method definition should be preceded by a member access modifier.

The three integer instance variables **hour**, **minute** and **second** are each declared (lines 8–10) with member access modifier **private**, indicating that these instance variables are accessible only to methods of the class. When a program creates (instantiates) an object of the class, such instance variables are encapsulated in the object and can be accessed only through methods of that object’s class (normally through the class’s **public** methods). Typically, instance variables are declared **private**, and methods are declared **public**. It is possible to have **private** methods and **public** data, as we will see later. The **private** methods are known as **utility methods** or **helper methods** because they can be called only by other methods of that class and are used to support the operation of those methods. Using **public** data is uncommon and is a dangerous programming practice.
Access methods can read or display data. Another common use for access methods is to test whether a condition is true or false—such methods are often called predicate methods. An example of a predicate method would be an isEmpty method for any container class—a class capable of holding many objects—such as a linked list, a stack or a queue (these data structures are discussed in depth in Chapter 19, Chapter 20 and Chapter 21). A program might test isEmpty before attempting to read another item from the container object. A program might test isFull before attempting to insert another item into the container object.

Class Time1 contains the following public methods—Time1 (lines 15–18), setTime (lines 22–27), toUniversalString (lines 30–37) and toString (line 40–48). These are the public methods, public services or public interface of the class. These methods are used by clients (i.e., portions of a program that are users of a class) of the class to manipulate the data stored in objects of the class.

The clients of a class use references to interact with an object of the class. For example, method paint in an applet is a client of class Graphics. Method paint uses a reference to a Graphics object (such as g) that it receives as an argument to draw on the applet by calling methods that are public services of class Graphics (such as drawString, drawLine, drawOval and drawRect).

Notice the method with the same name as the class (lines 15–18); it is the constructor method of that class. A constructor is a special method that initializes the instance variables of a class object. Java calls a class’s constructor method when a program instantiates an object of that class. In this example, the constructor simply calls the class’s setTime method (discussed shortly) with hour, minute and second values specified as 0.

It is common to have several constructors for a class; this is accomplished through method overloading (as we will see Fig. 8.6). Constructors can take arguments but cannot specify a return data type. Implicitly, the constructor returns a reference to the instantiated object. An important difference between constructors and other methods is that constructors are not allowed to specify a return data type (not even void). Normally, constructors are public methods of a class. Nonpublic methods are discussed later.

Method setTime (lines 22–27) is a public method that receives three integer arguments and uses them to set the time. Each argument is tested in a conditional expression that determines whether the value is in range. For example, the hour value must be greater than or equal to 0 and less than 24 because we represent the time in universal time format (0–23 for the hour, 0–59 for the minute and 0–59 for the second). Any value outside this
range is an invalid value and is set to zero—ensuring that a Time1 object always contains valid data. This is also known as keeping the object in a consistent state or maintaining the object’s integrity. In cases where invalid data is supplied to `setTime`, the program may want to indicate that an invalid time setting was attempted. We explore this possibility in the exercises.

**Good Programming Practice 8.1**

Always define a class so its instance variables are maintained in a consistent state.

Method `toUniversalString` (lines 30–37) takes no arguments and returns a `String`. This method produces a universal-time-format string consisting of six digits—two for the hour, two for the minute and two for the second. For example, 13:30:07 represents 1:30:07 PM. Line 32 creates an instance of class `DecimalFormat` (from package `java.text` imported at line 3) to help format the universal time. Object `twoDigits` is initialized with the format control string "00", which indicates that the number format should consist of two digits—each 0 is a placeholder for a digit. If the number being formatted is a single digit, it is automatically preceded by a leading 0 (i.e., 8 is formatted as 08). The return statement at lines 34–36 uses method `format` (which returns a formatted `String` containing the number) from object `twoDigits` to format the `hour`, `minute` and `second` values into two-digit strings. Those strings are concatenated with the + operator (separated by colons) and returned from method `toUniversalString`.

Method `toString` (line 40–48) takes no arguments and returns a `String`. This method produces a standard-time-format string consisting of the `hour`, `minute` and `second` values separated by colons and an AM or PM indicator, as in 1:27:06 PM. This method uses the same `DecimalFormat` techniques as method `toUniversalString` to guarantee that the `minute` and `second` values each appear with two digits. Method `toString` is special, in that we inherited from class `Object a toString` method with exactly the same first line as our `toString` on line 40. The original `toString` method of class `Object` is a generic version that is used mainly as a placeholder that can be redefined by a subclass (similar to methods `init`, `start` and `paint` from class `JApplet`). Our version replaces the version we inherited to provide a `toString` method that is more appropriate for our class. This is known as overriding the original method definition (discussed in detail in Chapter 9).

Once the class has been defined, it can be used as a type in declarations such as

```java
Time1 sunset,         // reference to object of type Time1
timeArray[];    // reference to array of Time1 objects
```

The class name is a new type specifier. There may be many objects of a class, just as there may be many variables of a primitive data type such as `int`. The programmer can create new class types as needed; this is one of the reasons that Java is known as an extensible language.

The `TimeTest1` application of Fig. 8.2 uses class `Time1`. Method `main` of class `TimeTest1` declares and initializes an instance of class `Time1` called `time` in line 12. When the object is instantiated, operator `new` allocates the memory in which the `Time1` object will be stored; then `new` calls the `Time1` constructor to initialize the instance variables of the new `Time1` object. The constructor invokes method `setTime` to explicitly initialize each private instance variable to 0. Operator `new` then returns a reference to the
new object, and that reference is assigned to `time`. Similarly, line 32 in class `Time1` (Fig. 8.1) uses `new` to allocate the memory for a `DecimalFormat` object, then calls the `DecimalFormat` constructor with the argument "00" to indicate the number format control string.

**Software Engineering Observation 8.5**

`new` creates an object of a class, that class’s constructor is called to initialize the instance variables of the new object.

Note that class `Time1` was not imported into the `TimeTest1.java` file. Actually, every class in Java is part of a package (like the classes from the Java API). If the programmer does not specify the package for a class, the class is automatically placed in the `default package`, which includes the compiled classes in the current directory. If a class is in the same package as the class that uses it, an `import` statement is not required. We import classes from the Java API because their `.class` files are not in the same package with each program we write. Section 8.5 illustrates how to define your own packages of classes for reuse.

```java
// Fig. 8.2: TimeTest1.java
// Class TimeTest1 to exercise class Time1

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

public class TimeTest1 {

    // create Time1 object and manipulate it
    public static void main( String args[] )
    {
        Time1 time = new Time1(); // calls Time1 constructor

        // append String version of time to String output
        String output = "The initial universal time is: " +
                        time.toUniversalString() +
                        "\nThe initial standard time is: " + time.toString() +
                        "\nImplicit toString() call: " + time;

        // change time and append String version of time to output
        time.setTime( 13, 27, 6 );
        output += "\nUniversal time after setTime is: " +
                  time.toUniversalString() +
                  "\nStandard time after setTime is: " + time.toString();

        // use invalid values to change time and append String
        // version of time to output
        time.setTime( 99, 99, 99 );
        output += "\n\nUniversal time: " + time.toUniversalString() +
                  "\nStandard time: " + time.toString();
    }
}
```

**Fig. 8.2** Using an object of class `Time1` in a program (part 1 of 2).
The statement at lines 15–18 defines a String reference output, which stores the string containing the results that will be displayed in a message box. Initially, the program assigns to output the time in universal-time format (by sending message toUniversalString to the object to which time refers) and standard-time format (by sending message toString to the object to which time refers) to confirm that the data were initialized properly. Note that line 18 uses a special string concatenation feature of Java. Concatenating a String with any object results in an implicit call to the object’s toString method to convert the object to a String; then the Strings are concatenated. Lines 17–18 illustrate that you can call toString both explicitly and implicitly in a String concatenation operation.

Line 21 sends the setTime message to the object to which time refers to change the time. Then lines 22–24 append the time to output again in both formats to confirm that the time was set correctly.

To illustrate that method setTime validates the values passed to it, line 28 calls method setTime and attempts to set the instance variables to invalid values. Then lines 29–31 append the time to output again in both formats to confirm that setTime validated the data. Lines 33–35 display a message box with the results of our program. Notice in the last two lines of the output window that the time is set to midnight—the default value of a Time1 object.

Now that we have seen our first non-applet, non-application class, let us consider several issues of class design.

Again, note that the instance variables hour, minute and second are each declared private. Instance variables declared private are not accessible outside the class in which they are defined. The philosophy here is that the actual data representation used within the class is of no concern to the class’s clients. For example, it would be perfectly

```java
JOptionPane.showMessageDialog(null, output,
    "Testing Class Time1",
    JOptionPane.INFORMATION_MESSAGE);

System.exit(0);

} // end class TimeTest1
```

Fig. 8.2 Using an object of class Time1 in a program (part 2 of 2).
reasonable for the class to represent the time internally as the number of seconds since midnight. Clients could use the same public methods and get the same results without being aware of this. In this sense, implementation of a class is said to be hidden from its clients. Exercise 8.18 asks you to make precisely this modification to the Time1 class of Figure 8.1 and show that there is no change visible to the clients of the class.

Software Engineering Observation 8.6

Information hiding promotes program modifiability and simplifies the client’s perception of a class. The client should not require knowledge of a class’s implementation (known as implementation knowledge) to be able to reuse a class.

Software Engineering Observation 8.7

Clients of a class can (and should) use the class without knowing the internal details of how the class is implemented. If the class implementation is changed (to improve performance, for example), the class clients’ source code need not change, provided that the class’s interface remains constant. This makes it much easier to modify systems.

In this program, the Time1 constructor simply initializes the instance variables to 0 (i.e., the universal time equivalent of 12 AM). This ensures that the object is created in a consistent state (i.e., all instance variable values are valid). Invalid values cannot be stored in the instance variables of a Time1 object, because the constructor is automatically called when the Time1 object is created and subsequent attempts by a client to modify the instance variables are scrutinized by the method setTime.

Instance variables can be initialized where they are declared in the class body, by the class’s constructor, or they can be assigned values by “set” methods. (Remember, instance variables that are not initialized explicitly receive default values (primitive numeric variables are set to 0, boolean values are set to false and references are set to null).

Good Programming Practice 8.2

Initialize instance variables of a class in that class’s constructor.

Every class may include a finalizer method called finalize that does “termination housekeeping” on each class object before the memory for the object is garbage collected by the system. We will discuss garbage collection and finalizers in detail in Section 8.14 and Section 8.15.

It is interesting that the toUniversalString and toString methods take no arguments. This is because these methods implicitly know that they are to manipulate the instance variables of the particular Time1 object for which they are invoked. This makes method calls more concise than conventional function calls in procedural programming. It also reduces the likelihood of passing the wrong arguments, the wrong types of arguments and/or the wrong number of arguments, as often happens in C function calls.

Software Engineering Observation 8.8

Using an object-oriented programming approach can often simplify method calls by reducing the number of parameters to be passed. This benefit of object-oriented programming derives from the fact that encapsulation of instance variables and methods within an object gives the methods the right to access the instance variables.

Classes simplify programming because the client (or user of the class object) need only be concerned with the public operations encapsulated in the object. Such operations are
usually designed to be client oriented rather than implementation oriented. Clients need not be concerned with a class’s implementation. Interfaces do change, but less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. By hiding the implementation, we eliminate the possibility of other program parts becoming dependent on the details of the class implementation.

Often, classes do not have to be created “from scratch.” Rather, they can be derived from other classes that provide operations the new classes can use, or classes can include objects of other classes as members. Such software reuse can greatly enhance programmer productivity. Deriving new classes from existing classes is called inheritance—a distinguishing feature between object-based programming and object-oriented programming—and is discussed in detail in Chapter 9. Including class objects as members of other classes is called composition or aggregation and is discussed later in this chapter.

### 8.3 Class Scope

A class’s instance variables and methods belong to that class’s scope. Within a class’s scope, class members are accessible to all of that class’s methods and can be referenced simply by name. Outside a class’s scope, class members cannot be referenced directly by name. Those class members (such as public members) that are visible can be accessed only through a “handle” (i.e., primitive data type members can be referred to by object-ReferenceName.primitiveVariableName and object members can be referenced by object-ReferenceName.objectMemberName). For example, a program can determine the number of elements in an array object by accessing the array’s public member length as in arrayName.length.

Variables defined in a method are known only to that method (i.e., they are local variables to that method). Such variables are said to have block scope. If a method defines a variable with the same name as a variable with class scope (i.e., an instance variable), the class-scope variable is hidden by the method-scope variable in the method scope. A hidden instance variable can be accessed in the method by preceding its name with the keyword this and the dot operator, as in this.variableName. Keyword this is discussed in Section 8.13.

### 8.4 Controlling Access to Members

The member access modifiers public and private control access to a class’s instance variables and methods. (In Chapter 9, we will introduce the additional access modifier protected.) As we stated previously, the primary purpose of public methods is to present to the class’s clients a view of the services the class provides (i.e., the public interface of the class). Clients of the class need not be concerned with how the class accomplishes its tasks. For this reason, the private instance variables and private methods of a class (i.e., the class’s implementation details) are not accessible to the clients of a class. Restricting access to class members via keyword private is called encapsulation.

**Common Programming Error 8.3**

An attempt by a method that is not a member of a particular class to access a private member of that class is a syntax error.

Figure 8.3 demonstrates that private class members are not accessible by name outside the class. Line 10 attempts to access directly the private instance variable hour of
the `Time1` object to which `time` refers. When this program is compiled, the compiler generates an error stating that the `private` member `hour` is not accessible. [Note: This program assumes that the `Time1` class from Figure 8.1 is used.]

**Good Programming Practice 8.3**

Our preference is to list the `private` instance variables of a class first, so that, as you read the code, you see the names and types of the instance variables before they are used in the methods of the class.

**Software Engineering Observation 8.9**

Keep all the instance variables of a class `private`. When necessary, provide `public` methods to set the values of `private` instance variables and to get the values of `private` instance variables. This architecture helps hide the implementation of a class from its clients, which reduces bugs and improves program modifiability.

Access to `private` data should be controlled carefully by the class’s methods. For example, to allow clients to read the value of `private` data, the class can provide a “get” method (also called an accessor method). To enable clients to modify `private` data, the class can provide a “set” method (also called a mutator method). Such modification would seem to violate the notion of `private` data, but a set method can provide data validation capabilities (such as range checking) to ensure that the value is set properly. A set method can also translate between the form of the data used in the interface and the form used in the implementation. A get method need not expose the data in “raw” format; rather, the get method can edit the data and limit the view of the data the client will see.

**Software Engineering Observation 8.10**

Class designers use `private` data and `public` methods to enforce the notion of information hiding and the principle of least privilege. If the client of a class needs access to data in the class, provide that access through `public` methods of the class. By doing so, the programmer of the class controls how the class’s data is manipulated (e.g., data validity checking can prevent invalid data from being stored in an object). This is the principle of data encapsulation.

```java
// Fig. 8.3: TimeTest2.java
// Demonstrate errors resulting from attempts
// to access private members of class Time1.
public class TimeTest2 {

    public static void main( String args[] )
    {
        Time1 time = new Time1();
        time.hour = 7;
    }
}
```

Fig. 8.3  Erroneous attempt to access private members of class `Time1`. © Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
Software Engineering Observation 8.11
The class designer need not provide set and/or get methods for each private data member; these capabilities should be provided only when it makes sense and after careful thought by the class designer.

Testing and Debugging Tip 8.1
Making the instance variables of a class private and the methods of the class public facilitates debugging because problems with data manipulations are localized to the class’s methods.

8.5 Creating Packages
As we have seen in almost every example in the text, classes and interfaces (discussed in Chapter 9) from preexisting libraries, such as the Java API, can be imported into a Java program. Each class and interface in the Java API belongs to a specific package that contains a group of related classes and interfaces. As applications become more complex, packages help programmers manage the complexity of application components. Packages also facilitate software reuse by enabling programs to import classes from other packages (as we have done in almost every example to this point). Another benefit of packages is that they provide a convention for unique class names. With hundreds of thousands of Java programmers around the world, there is a good chance that the names you choose for classes will conflict with the names that other programmers choose for their classes. This section introduces how to create your own packages and discusses the standard distribution mechanism for packages.

The application of Fig. 8.4 and Fig. 8.5 illustrates how to create your own package and use a class from that package in a program. The steps for creating a reusable class are:

1. Define a public class. If the class is not public, it can be used only by other classes in the same package.

2. Choose a package name, and add a package statement to the source code file for the reusable class definition. [Note: There can be only one package statement in a Java source code file.]

3. Compile the class so it is placed in the appropriate package directory structure.

4. Import the reusable class into a program, and use the class.

Common Programming Error 8.4
A syntax error occurs if any code appears in a Java file before the package statement (if there is one) in the file.

We chose to demonstrate Step 1 by modifying the public class Time1 defined in Fig. 8.1. The new version is shown in Fig. 8.4. No modifications have been made to the implementation of the class, so we will not discuss the implementation details of the class again here.

To satisfy Step 2, we added a package statement at the beginning of the file. Line 3 uses a package statement to define a package named com.deitel.jhttp4.ch08. Placing a package statement at the beginning of a Java source file indicates that the class defined in the file is part of the specified package. The only types of statements in Java that can appear outside the braces of a class definition are package statements and import statements.
public class Time1 extends Object {
  private int hour;     // 0 - 23
  private int minute;   // 0 - 59
  private int second;   // 0 - 59

  // Time1 constructor initializes each instance variable
  // to zero. Ensures that each Time1 object starts in a
  // consistent state.
  public Time1()
  {
    setTime( 0, 0, 0 );
  }

  // Set a new time value using universal time. Perform
  // validity checks on the data. Set invalid values to zero.
  public void setTime( int h, int m, int s )
  {
    hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
    second = ( ( s >= 0 && s < 60 ) ? s : 0 );
  }

  // convert to String in universal-time format
  public String toUniversalString()
  {
    DecimalFormat twoDigits = new DecimalFormat( "00" );
    return twoDigits.format( hour ) + ":" +
            twoDigits.format( minute ) + ":" +
            twoDigits.format( second );
  }

  // convert to String in standard-time format
  public String toString()
  {
    DecimalFormat twoDigits = new DecimalFormat( "00" );
    return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
            ":" + twoDigits.format( minute ) +
            ":" + twoDigits.format( second ) +
            ( hour < 12 ? " AM" : " PM" );
  }
} // end class Time1
Software Engineering Observation 8.12

A Java source code file has the following order: a package statement (if any), zero or more import statements, then class definitions. Only one of the class definitions in a particular file can be public. Other classes in the file are also placed in the package, but are reusable only from other classes in that package—they cannot be imported into classes in another package. They are in the package to support the reusable class in the file.

In an effort to provide unique names for every package, Sun Microsystems specifies a convention for package naming that all Java programmers should follow. Every package name should start with your Internet domain name in reverse order. For example, our Internet domain name is deitel.com, so we began our package name with com.deitel. If your domain name is yourcollege.edu, the package name you would use is edu.yourcollege. After the domain name is reversed, you can choose any other names you want for your package. If you are part of a company with many divisions or a university with many schools, you may want to use the name of your division or school as the next name in the package. We chose to use jhtp4 as the next name in our package name to indicate that this class is from Java How to Program: Fourth Edition. The last name in our package name specifies that this package is for Chapter 8 (ch08). [Note: We use our own packages several times throughout the book. You can determine the chapter in which one of our reusable classes is defined by looking at the last part of the package name in the import statement. This appears before the name of the class being imported or before the * if a particular class is not specified.]

Step 3 is to compile the class so it is stored in the appropriate package. When a Java file containing a package statement is compiled, the resulting .class file is placed in the directory structure specified by the package statement. The preceding package statement indicates that class Time1 should be placed in the directory ch08. The other names—com, deitel and jhtp4—are also directories. The directory names in the package statement specify the exact location of the classes in the package. If these directories do not exist before the class is compiled, the compiler creates them.

When compiling a class in a package, there is an extra option (-d) that must be passed to the javac compiler. This option specifies where to create (or locate) the directories in the package statement. For example, we used the compilation command

    javac -d . Time1.java

to specify that the first directory specified in our package name should be placed in the current directory. The . after -d in the preceding command represents the current directory on the Windows, UNIX and Linux operating systems (and several others as well). After executing the compilation command, the current directory contains a directory called com. com contains a directory called deitel.deitel contains a directory called jhtp4 and jhtp4 contains a directory called ch08. In the ch08 directory, you can find the file Time1.class.

The package directory names become part of the class name when the class is compiled. The class name in this example is actually com.deitel.jhtp4.ch08.Time1 after the class is compiled. You can use this fully qualified name in your programs or you can import the class and use its short name (Time1) in the program. If another package also contains a Time1 class, the fully qualified class names can be used to distinguish between the classes in the program and prevent a naming conflict (also called a name collision).
Once the class is compiled and stored in its package, the class can be imported into programs (Step 4). The TimeTest3 application of Fig. 8.5, line 8 specifies that class Time1 should be imported for use in class TimeTest3.

At compile time for class TimeTest3, javac must locate .class file for Time1, so javac can ensure that class TimeTest3 uses class Time1 correctly. The compiler follows a specific search order to locate the classes it needs. It begins by searching the standard Java classes that are bundled with the J2SDK. Then it searches for extension classes. Java 2 provides an extensions mechanism that enables new packages to be added to Java for development and execution purposes. [Note: The extensions mechanism is beyond the scope of this book. For more information, visit java.sun.com/j2se/1.3/docs/guide/extensions.] If the class is not found in the standard Java classes or in the extension classes, the compiler searches the class path. By default, the class path consists only of the current directory. However, the class path can be modified by:

```java
// Fig. 8.5: TimeTest3.java
// Class TimeTest3 to use imported class Time1

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

// Deitel packages
import com.deitel.jhtp4.ch08.Time1; // import Time1 class

public class TimeTest3 {

    // create an object of class Time1 and manipulate it
    public static void main( String args[] )
    {
        Time1 time = new Time1(); // create Time1 object

        time.setTime( 13, 27, 06 ); // set new time
        String output =
            "Universal time is: " + time.toUniversalString() +
            "\nStandard time is: " + time.toString();

        JOptionPane.showMessageDialog( null, output,
            "Packaging Class Time1 for Reuse",
            JOptionPane.INFORMATION_MESSAGE);

        System.exit( 0 );
    }
}
```

Fig. 8.5 Using programmer-defined class Time1 in a package.
1. providing the `-classpath` option to the `javac` compiler, or
2. setting the `CLASSPATH` environment variable (a special variable that you define and the operating system maintains so that applications can search for classes in the specified locations).

In each case, the class path consists of a list of directories and/or archive files separated by semicolons (`;`). Archive files are individual files that contain directories of other files, typically in compressed format. For example, the standard classes of Java are contained in the archive file `rt.jar` that is installed with the J2SDK. Archive files normally end with the `.jar` or `.zip` file name extensions. The directories and archive files specified in the class path contain the classes you wish to make available to the Java compiler. For more information on the class path, visit `java.sun.com/j2se/1.3/docs/tooldocs/win32/classpath.html` for Windows or `java.sun.com/j2se/1.3/docs/tooldocs/solaris/classpath.html` for Solaris/Linux. [Note: We discuss archive files in more detail in Section 8.8.]

### Common Programming Error 8.5
Specifying an explicit class path eliminates the current directory from the class path. This prevents classes in the current directory from loading properly. If classes must be loaded from the current directory, include the current directory (`.`) in the explicit class path.

### Software Engineering Observation 8.13
In general, it is a better practice to use the `-classpath` option of the compiler, rather than the `CLASSPATH` environment variable, to specify the class path for a program. This enables each application to have its own class path.

### Testing and Debugging Tip 8.2
Specifying the class path with the `CLASSPATH` environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same packages.

For the example of Fig. 8.4 and Fig. 8.5, we did not specify an explicit class path. Thus, to locate the classes in the `com.deitel.jhtp4.ch08` package from this example, the compiler looks in the current directory for the first name in the package—`com`. Next, the compiler navigates the directory structure. Directory `com` contains the subdirectory `deitel`. Directory `deitel` contains the subdirectory `jhtp4`. Finally, directory `jhtp4` contains subdirectory `ch08`. In the `ch08` directory is the file `Time1.class`, which is loaded by the compiler to ensure that the class is used properly in our program.

Locating the classes to execute the program is similar to locating the classes to compile the program. Like the compiler, the `java` interpreter searches the standard classes and extension classes first, then searches the class path (the current directory by default). The class path for the interpreter can be specified explicitly by using either of the techniques discussed for the compiler. As with the compiler, it is better to specify an individual program’s class path via command-line options to the interpreter. You can specify the class path to the `java` interpreter via the `-classpath` or `-cp` command line options followed by a list of directories and/or archive files separated by semicolons (`;`).

### 8.6 Initializing Class Objects: Constructors

When an object is created, its members can be initialized by a `constructor` method. A constructor is a method with the same name as the class (including case sensitivity). The pro-
grammer of a class can define the class’s constructor, which is invoked each time the program instantiates an object of that class. Instance variables can be initialized implicitly to their default values (0 for primitive numeric types, false for booleans and null for references), they can be initialized in a constructor of the class or their values can be set later after the object is created. Constructors cannot specify return types or return values. A class can contain overloaded constructors to provide a variety of means for initializing objects of that class.

When a program instantiates an object of a class, the program can supply initializers in parentheses to the right of the class name. These initializers are passed as arguments to the class’s constructor. This technique is demonstrated in the example of Section 8.7. We have also seen this technique several times previously as we created new objects of classes like DecimalFormat, JLabel, JTextField, JTextArea and JButton. For each of these classes, we have seen statements of the form

```java
ref = new ClassName( arguments );
```

where ref is a reference of the appropriate data type, new indicates that a new object is being created, ClassName indicates the type of the new object and arguments specifies the values used by the class’s constructor to initialize the object.

If no constructors are defined for a class, the compiler creates a default constructor that takes no arguments (also called a no-argument constructor). The default constructor for a class calls the default constructor for its superclass (the class it extends), then proceeds to initialize the instance variables in the manner we discussed previously. If the class that this class extends does not have a default constructor, the compiler issues an error message. It is also possible for the programmer to provide a no-argument constructor, as we showed in class Time1 and will see in the next example. If any constructors are defined for a class by the programmer, Java will not create a default constructor for the class.

**Good Programming Practice 8.4**

When appropriate (almost always), provide a constructor to ensure that every object is properly initialized with meaningful values.

**Common Programming Error 8.6**

If constructors are provided for a class, but none of the public constructors are no-argument constructors, and an attempt is made to call a no-argument constructor to initialize an object of the class, a syntax error occurs. A constructor can be called with no arguments only if there are no constructors for the class (the default constructor is called) or if there is a no-argument constructor.

### 8.7 Using Overloaded Constructors

Methods of a class can be overloaded (i.e., several methods in a class may have exactly the same name as defined in Chapter 6, Methods). To overload a method of a class, simply provide a separate method definition with the same name for each version of the method. Remember that overloaded methods must have different parameter lists.

**Common Programming Error 8.7**

Attempting to overload a method of a class with another method that has the exact same signature (name and parameters) is a syntax error.

The Time1 constructor in Fig. 8.1 initialized hour, minute and second to 0 (i.e., 12 midnight in universal time) with a call to the class’s setTimex method. Figure 8.6
overloads the constructor method to provide a convenient variety of ways to initialize objects of the new class `Time2`. The constructors guarantee that every object begins its existence in a consistent state. In this program, each constructor calls method `setTime` with the values passed to the constructor, to ensure that the value supplied for `hour` is in the range 0 to 23 and that the values for `minute` and `second` are each in the range 0 to 59. If a value is out of range, it is set to zero by `setTime` (once again ensuring that each instance variable remains in a consistent state). The appropriate constructor is invoked by matching the number, types and order of the arguments specified in the constructor call with the number, types and order of the parameters specified in each constructor definition. The matching constructor is called automatically. Figure 8.7 uses class `Time2` to demonstrate its constructors.

```java
package com.deitel.jhtp4.ch08;

import java.text.DecimalFormat;

public class Time2 extends Object {
    private int hour;     // 0 - 23
    private int minute;   // 0 - 59
    private int second;   // 0 - 59

    // Time2 constructor initializes each instance variable
    // to zero. Ensures that Time object starts in a
    // consistent state.
    public Time2()
    {
        setTime( 0, 0, 0 );
    }

    // Time2 constructor: hour supplied, minute and second
    // defaulted to 0
    public Time2( int h )
    {
        setTime( h, 0, 0 );
    }

    // Time2 constructor: hour and minute supplied, second
    // defaulted to 0
    public Time2( int h, int m )
    {
        setTime( h, m, 0 );
    }

    // Time2 constructor: hour, minute and second supplied
    public Time2( int h, int m, int s )
    {
        setTime( h, m, s );
    }
}
```

Fig. 8.6   Class `Time2` with overloaded constructors (part 1 of 2).
Most of the code in class `Time2` is identical to that in class `Time1`, so we concentrate on only the new features here (i.e., the constructors). Lines 16–19 define the no-argument (default) constructor. Lines 23–26 define a `Time2` constructor that receives a single `int` argument representing the `hour`. Lines 30–33 defines a `Time2` constructor that receives two `int` arguments representing the `hour` and `minute`. Lines 36–39 define a `Time2` constructor that receives three `int` arguments representing the `hour`, `minute` and `second`. Lines 42–45 define a `Time2` constructor that receives a `Time2` reference to another `Time2` object. In this case, the values from the `Time2` argument are used to initialize the `hour`, `minute` and `second`. Notice that none of the constructors specifies a return data type (remember, this is not allowed for constructors). Also, notice that all the constructors receive different numbers of arguments and/or different types of arguments. Even though only two
of the constructors receive values for the `hour`, `minute` and `second`, all the constructors call `setTime` with values for `hour`, `minute` and `second` and substitute zeros for the missing values to satisfy `setTime`'s requirement of three arguments.

Notice in particular the constructor at lines 42–45, which uses the `hour`, `minute` and `second` values of its argument `time` to initialize the new `Time2` object. Even though we know that `hour`, `minute` and `second` are declared as `private` variables of class `Time2`, we are able to access these values directly by using the expressions `time.hour`, `time.minute` and `time.second`. This is due to a special relationship between objects of the same class. When one object of a class has a reference to another object of the same class, the first object can access all the second object’s data and methods.

Class `TimeTest4` (Fig. 8.7) creates six `Time2` objects (lines 17–22) to demonstrate how to invoke the different constructors of the class. Line 17 shows that the no-argument constructor is invoked by placing an empty set of parentheses after the class name when allocating a `Time2` object with `new`. Lines 18–22 of the program demonstrate passing arguments to the `Time2` constructors. Remember that the appropriate constructor is invoked by matching the number, types and order of the arguments specified in the constructor call with the number, types and order of the parameters specified in each method definition. So, line 18 invokes the constructor at line 23 of Fig. 8.6. Line 19 invokes the constructor at line 30 of Fig. 8.6. Lines 20–21 invoke the constructor at line 36 of Fig. 8.6. Line 22 invokes the constructor at line 42 of Fig. 8.6.

Note that each `Time2` constructor in Fig. 8.6 could be written to include a copy of the appropriate statements from method `setTime`. This could be slightly more efficient, because the extra call to `setTime` is eliminated. However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed in the day (requiring 4 bytes of memory). Coding the `Time2` constructors and method `setTime` identically makes such a change in this class definition more difficult. If the implementation of method `setTime` changes, the implementation of the `Time2` constructors would need to change accordingly. Having the `Time2` constructors call `setTime` directly requires any changes to the implementation of `setTime` to be made only once. This reduces the likelihood of a programming error when altering the implementation.

**Software Engineering Observation 8.14**

If a method of a class already provides all or part of the functionality required by a constructor (or other method) of the class, call that method from the constructor (or other method). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. It is also an effective example of reuse.
public class TimeTest4 {
    // test constructors of class Time2
    public static void main( String args[] )
    {
        Time2 t1, t2, t3, t4, t5, t6;

        t1 = new Time2();               // 00:00:00
        t2 = new Time2( 2 );            // 02:00:00
        t3 = new Time2( 21, 34 );       // 21:34:00
        t4 = new Time2( 12, 25, 42 );   // 12:25:42
        t5 = new Time2( 27, 74, 99 );   // 00:00:00
        t6 = new Time2( t4 );           // 12:25:42

        String output = "Constructed with: "+
                         "\nt1: all arguments defaulted" +
                         "\n" + t1.toUniversalString() + 
                         "\n" + t1.toString();

        output += "\nt2: hour specified; minute and " +
                  "second defaulted" +
                  "\n" + t2.toUniversalString() +
                  "\n" + t2.toString();

        output += "\nt3: hour and minute specified; " +
                  "second defaulted" +
                  "\n" + t3.toUniversalString() +
                  "\n" + t3.toString();

        output += "\nt4: hour, minute, and second specified" +
                  "\n" + t4.toUniversalString() +
                  "\n" + t4.toString();

        output += "\nt5: all invalid values specified" +
                  "\n" + t5.toUniversalString() +
                  "\n" + t5.toString();

        output += "\nt6: Time2 object t4 specified" +
                  "\n" + t6.toUniversalString() +
                  "\n" + t6.toString();

        JOptionPane.showMessageDialog( null, output,
                                       "Demonstrating Overloaded Constructors",
                                       JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }
} // end class TimeTest4

Fig. 8.7 Using overloaded constructors to initialize objects of class Time2 (part 2 of 3).
8.8 Using Set and Get Methods

Private instance variables can be manipulated only by methods of the class. A typical manipulation might be the adjustment of a customer’s bank balance (e.g., a private instance variable of a class BankAccount) by a method computeInterest.

Classes often provide public methods to allow clients of the class to set (i.e., assign values to) or get (i.e., obtain the values of) private instance variables. These methods need not be called set and get, but they often are.

As a naming example, a method that sets instance variable interestRate would typically be named setInterestRate and a method that gets the interestRate would typically be called getInterestRate. Get methods are also commonly called accessor methods or query methods. Set methods are also commonly called mutator methods (because they typically change a value).

It would seem that providing set and get capabilities is essentially the same as making the instance variables public. This is another subtlety of Java that makes the language so desirable for software engineering. If an instance variable is public, the instance variable can be read or written at will by any method in the program. If an instance variable is private, a public get method certainly seems to allow other methods to read the data at will but the get method controls the formatting and display of the data. A public set method can—and most likely will—carefully scrutinize attempts to modify the instance variable’s value. This ensures that the new value is appropriate for that data item. For example, an attempt to set the day of the month for a date to 37 would be rejected, an attempt to set a person’s weight to a negative value would be rejected, and so on. So,
although *set* and *get* methods could provide access to *private* data, the access is restricted by the programmer’s implementation of the methods.

The benefits of data integrity are not automatic simply because instance variables are made *private*—the programmer must provide validity checking. Java provides the framework in which programmers can design better programs in a convenient manner.

**Software Engineering Observation 8.15**

Methods that set the values of *private* data should verify that the intended new values are proper; if they are not, the set methods should place the *private* instance variables into an appropriate consistent state.

A class’s *set* methods can return values indicating that attempts were made to assign invalid data to objects of the class. This enables clients of the class to test the return values of *set* methods to determine whether the objects they are manipulating are valid and to take appropriate action if the objects are not valid. In Chapter 14, Exception Handling, we illustrate a more robust way in which clients of a class can be notified if an object is not valid.

**Good Programming Practice 8.5**

Every method that modifies the *private* instance variables of an object should ensure that the data remains in a consistent state.

The applet of Fig. 8.8 (class *Time3*) and Fig. 8.9 (class *TimeTest5*) enhances our *Time* class (now called *Time3*) to include *get* and *set* methods for the *hour*, *minute*, and *second* *private* instance variables. The *set* methods strictly control the setting of the instance variables to valid values. Attempts to set any instance variable to an incorrect value cause the instance variable to be set to zero (thus leaving the instance variable in a consistent state). Each *get* method simply returns the appropriate instance variable’s value. This applet introduces enhanced GUI event handling techniques as we move toward defining our first full-fledged windowed application. After discussing the code, we introduce how to set up an applet to use classes in programmer-defined packages.
// Time3 constructor: hour supplied, minute and second
// defaulted to 0
public Time3( int h )
{
    setTime( h, 0, 0 );
}

// Time3 constructor: hour and minute supplied, second
// defaulted to 0
public Time3( int h, int m )
{
    setTime( h, m, 0 );
}

// Time3 constructor: hour, minute and second supplied
public Time3( int h, int m, int s )
{
    setTime( h, m, s );
}

// Time3 constructor: another Time3 object supplied
public Time3( Time3 time )
{
    setTime( time.getHour(), time.getMinute(),
             time.getSecond() );
}

// Set Methods
// Set a new time value using universal time. Perform
// validity checks on data. Set invalid values to zero.
public void setTime( int h, int m, int s )
{
    setHour( h );    // set the hour
    setMinute( m );  // set the minute
    setSecond( s );  // set the second
}

// validate and set hour
public void setHour( int h )
{
    hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
}

// validate and set minute
public void setMinute( int m )
{
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
}
The new set methods of the class are defined in Fig. 8.8 at lines 59–62, 65–68 and 71–74, respectively. Notice that each method performs the same conditional statement that was
previously in method `setTime` for setting the `hour`, `minute` or `second`. The addition of these methods caused us to redefine the body of method `setTime` to follow *Software Engineering Observation 8.14*—if a method of a class already provides all or part of the functionality required by another method of the class, call that method from the other method. Notice that `setTime` (lines 51–56) now calls methods `setHour`, `setMinute` and `setSecond`—each of which performs part of `setTime`’s task.

The new `get` methods of the class are defined at lines 78–81, 84–87 and 90–93, respectively. Notice that each method simply returns the `hour`, `minute` or `second` value (a copy of each value is returned, because these are all primitive data type variables). The addition of these methods caused us to redefine the bodies of methods `toUniversalString` (lines 96–103) and `toString` (lines 106–115) to follow *Software Engineering Observation 8.14*. In both cases, every use of instance variables `hour`, `minute` and `second` is replaced with a call to `getHour`, `getMinute` and `getSecond`.

Due to the changes in class `Time3` just described, we have minimized the changes that will have to occur in the class definition if the data representation is changed from `hour`, `minute` and `second` to another representation (such as total elapsed seconds in the day). Only the new `set` and `get` method bodies will have to change. This allows the programmer to change the implementation of the class without affecting the clients of the class (as long as all the `public` methods of the class are still called the same way).

The `TimeTest5` applet (Fig. 8.9) provides a graphical user interface that enables the user to exercise the methods of class `Time3`. The user can set the hour, minute or second value by typing a value in the appropriate `JTextField` and pressing the `Enter` key. The user can also click the “Add 1 to second” button to increment the time by one second. The `JTextField` and `JButton` events in this applet are all processed in method `actionPerformed` (lines 66–94). Notice that lines 34, 41, 48 and 59 all call `addActionListener` to indicate that the applet should start listening to `JTextField`'s `hourField`, `minuteField`, `secondField` and `JButton` `tickButton`, respectively. Also, notice that all four calls use `this` as the argument, indicating that the object of our applet class `TimeTest5` has its `actionPerformed` method invoked for each user interaction with these four GUI components. This poses an interesting question—how do we determine the GUI component with which the user interacted?

In `actionPerformed`, notice the use of `actionEvent.getSource()` to determine which GUI component generated the event. For example, line 69 determines whether `tickButton` was clicked by the user. If so, the body of the `if` structure executes. Otherwise, the program tests the condition in the `if` structure at line 73, and so on. Every event has a `source`—the GUI component with which the user interacted to signal the program to do a task. The `ActionEvent` parameter contains a reference to the source of the event. The condition in line 69 simply asks, “Is the `source` of the event the `tickButton`?” This condition compares the references on either side of the `==` operator to determine whether they refer to the same object. In this case, if they both refer to the `JButton`, then the program knows that the user pressed the button. Remember that the source of the event calls `actionPerformed` in response to the user interaction.

After each operation, the resulting time is displayed as a string in the status bar of the applet. The output windows in Fig. 8.9 illustrate the applet before and after the following operations: setting the hour to 23, setting the minute to 59, setting the second to 58 and incrementing the second twice with the “Add 1 to second” button.
// Fig. 8.9: TimeTest5.java
// Demonstrating the Time3 class set and get methods.

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

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

// Deitel packages
import com.deitel.jhtp4.ch08.Time3;

public class TimeTest5 extends JApplet
    implements ActionListener {
    private Time3 time;
    private JLabel hourLabel, minuteLabel, secondLabel;
    private JTextField hourField, minuteField, secondField, displayField;
    private JButton tickButton;

    // Create Time3 object and set up GUI
    public void init() {
        time = new Time3();
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // set up hourLabel and hourField
        hourLabel = new JLabel( "Set Hour" );
        hourField = new JTextField( 10 );
        hourField.addActionListener( this );
        container.add( hourLabel );
        container.add( hourField );

        // set up minuteLabel and minuteField
        minuteLabel = new JLabel( "Set minute" );
        minuteField = new JTextField( 10 );
        minuteField.addActionListener( this );
        container.add( minuteLabel );
        container.add( minuteField );

        // set up secondLabel and secondField
        secondLabel = new JLabel( "Set Second" );
        secondField = new JTextField( 10 );
        secondField.addActionListener( this );
        container.add( secondLabel );
        container.add( secondField );

        // set up displayField
        displayField = new JTextField( 30 );
    }
}
```java
displayField.setEditable( false );
container.add( displayField );

// set up tickButton
tickButton = new JButton( "Add 1 to Second" );
tickButton.addActionListener( this );
container.add( tickButton );

updateDisplay();  // update text in displayField

// handle button and text field events
public void actionPerformed( ActionEvent actionEvent )
{
    // process tickButton event
    if ( actionEvent.getSource() == tickButton )
        tick();

    // process hourField event
    else if ( actionEvent.getSource() == hourField ) {
        time.setHour(  
            Integer.parseInt( actionEvent.getActionCommand() )  
        );
        hourField.setText( "" );
    }

    // process minuteField event
    else if ( actionEvent.getSource() == minuteField ) {
        time.setMinute(  
            Integer.parseInt( actionEvent.getActionCommand() )  
        );
        minuteField.setText( "" );
    }

    // process secondField event
    else if ( actionEvent.getSource() == secondField ) {
        time.setSecond(  
            Integer.parseInt( actionEvent.getActionCommand() )  
        );
        secondField.setText( "" );
    }

    updateDisplay();  // update displayField and status bar
}

// update displayField and applet container's status bar
public void updateDisplay()
{
    displayField.setText( "Hour: " + time.getHour() +  
        "; Minute: " + time.getMinute() +  
        "; Second: " + time.getSecond() );

    showStatus( "Standard time is: " + time.toString() +  
        "; Universal time is: " + time.toUniversalString() );
}
```

Fig. 8.9 Using class Time3's set and get methods (part 2 of 4).
107 // add one to second and update hour/minute if necessary
108 public void tick()
109 {
110     time.setSecond( ( time.getSecond() + 1 ) % 60 );
111
112     if ( time.getSecond() == 0 ) {
113         time.setMinute( ( time.getMinute() + 1 ) % 60 );
114
115             if ( time.getMinute() == 0 )
116                 time.setHour( ( time.getHour() + 1 ) % 24 );
117     }
118 }
119 }
120 } // end class TimeTest5
Note that when the "Add 1 to second" button is clicked, method actionPerformed calls the applet’s tick method (defined at lines 108–118). Method tick uses all the new set and get methods to increment the second properly. Although this works, it incurs the performance burden of issuing multiple method calls. In Section 8.12, we discuss the notion of package access as a means of eliminating this performance burden.

**Common Programming Error 8.8**

A constructor can call other methods of the class, such as set or get methods. However, the instance variables might not yet be in a consistent state, because the constructor is in the process of initializing the object. Using instance variables before they have been initialized properly is a logic error.
Set methods are certainly important from a software engineering standpoint, because they can perform validity checking. Set and get methods have another important software engineering advantage, discussed in the following Software Engineering Observation.

**Software Engineering Observation 8.16**

Accessing private data through set and get methods not only protects the instance variables from receiving invalid values, but also insulates clients of the class from the representation of the instance variables. Thus, if the representation of the data changes (typically, to reduce the amount of storage required, improve performance or enhance the class in other ways), only the method implementations need to change—the clients need not change as long as the interface provided by the methods remains the same.

### 8.8.1 Executing an Applet that Uses Programmer-Defined Packages

After compiling the classes in Fig. 8.8 and Fig. 8.9, you can execute the applet from a command window with the command

```
appletviewer TimeTest5.html
```

As we discussed when we introduced packages earlier in this chapter, the interpreter can locate packaged classes in the current directory. The `appletviewer` is a Java application that executes a Java applet. Like the interpreter, the `appletviewer` can load standard Java classes and extension classes installed on the local computer. However, the `appletviewer` does not use the class path to locate classes in programmer-defined packages. For an applet, such classes should be bundled with the applet class in an archive file called a Java Archive (JAR) file. Remember that applets normally are downloaded from the Internet into a Web browser (see Chapter 3 for more information). Bundling the classes and packages that compose an applet enables the applet and its supporting classes to be downloaded as a unit, then executed in the browser (or via the Java Plug-in for browsers that do not support Java 2).

To bundle the classes in Fig. 8.8 and Fig. 8.9, open a command window and change directories to the location in which `TimeTest5.class` is stored. In that same directory should be the `com` directory that begins the package directory structure for class `Time3`. In that directory, issue the following command

```
jar cf TimeTest5.jar TimeTest5.class com\*.*
```

To create the JAR file. [*Note: This command uses \ as the directory separator from the MS-DOS prompt. UNIX would use / as the directory separator.] In the preceding command, `jar` is the Java archive utility used to create JAR files. Next are the options for the `jar` utility—`cf`. The letter `c` indicates that we are creating a JAR file. The letter `f` indicates that the next argument in the command line (`TimeTest5.jar`) is the name of the JAR file to create. Following the options and JAR file name are the actual files that will be included in the JAR file. We specified `TimeTest5.class` and `com\*.*`, indicating that `TimeTest5.class` and all the files in the `com` directory should be included in the JAR file. The `com` directory begins the package that contains the `.class` file for the `Time3`. [*Note: You can include selected files by specifying the path and file name for each individ-
ual file.] It is important that the directory structure in the JAR file match the directory structure for the packaged classes. Therefore, we executed the `jar` command from the directory in which `com` is located.

To confirm that the files were archived directly, you can issue the command

```
jar tvf TimeTest5.jar
```

which produces the listing in Fig. 8.10. In the preceding command, the options for the `jar` utility are `tvf`. The letter `t` indicates that the table of contents for the JAR should be listed. 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 to use.

The only remaining issue is to specify the archive as part of the applet’s HTML file. In prior examples, `<applet>` tags had the form

```
<applet code = "ClassName.class" width = "width" height = "height">
</applet>
```

To specify that the applet classes are located in a JAR file, use an `<applet>` tag of the form:

```
<applet code = "ClassName.class" archive = "archiveList"
   width = "width" height = "height">
</applet>
```

The `archive` attribute can specify a comma-separated list of archive files for use in an applet. Each file in the `archive` list will be downloaded by the browser when it encounters the `<applet>` tags in the HTML document. For the TimeTest5 applet, the applet tag would be

```
<applet code = "TimeTest5.class" archive = "TimeTest5.jar"
   width = "400" height = "115">
</applet>
```

Try loading this applet into your Web browser. Remember that you must either have a browser that supports Java 2 (such as Netscape Navigator 6) or convert the HTML file for use with the Java Plug-in (as discussed in Chapter 3).

---

**Fig. 8.10** Contents of `TimeTest5.jar`.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01
8.9 Software Reusability

Java programmers concentrate on crafting new classes and reusing existing classes. Many class libraries exist, and others are being developed worldwide. Software is then constructed from existing, well-defined, carefully tested, well-documented, portable, widely available components. This kind of software reusability speeds the development of powerful, high-quality software. Rapid applications development (RAD) is of great interest today.

Java programmers now have thousands of classes in the Java API from which to choose to help them implement Java programs. Indeed, Java is not just a programming language. It is a framework in which Java developers can work to achieve true reusability and rapid applications development. Java programmers can focus on the task at hand when developing their programs and leave the lower-level details to the classes of the Java API. For example, to write a program that draws graphics, a Java programmer does not require knowledge of graphics on every computer platform where the program will execute. Instead, a Java programmer concentrates on learning Java’s graphics capabilities (which are quite substantial and growing) and writes a Java program that draws the graphics, using Java’s API classes such as Graphics. When the program executes on a given computer, it is the job of the interpreter to translate Java commands into commands that the local computer can understand.

The Java API classes enable Java programmers to bring new applications to market faster by using preexisting, tested components. Not only does this reduce development time, it also improves programmers’ ability to debug and maintain applications. To take advantage of Java’s many capabilities, it is essential that programmers take the time to familiarize themselves with the variety of packages and classes in the Java API. There are many Web-based resources at java.sun.com to help you with this task. The primary resource for learning about the Java API is the Java API documentation, which can be found at

java.sun.com/j2se/1.3/docs/api/index.html

In addition, java.sun.com provides many other resources, including tutorials, articles and sites specific to individual Java topics. Java developers should also register (for free) at the Java Developer Connection

developer.java.sun.com

This site provides additional resources that Java developers will find useful, including more tutorials and articles, and links to other Java resources. See Appendix B for a more complete list of Java-related Internet and World Wide Web resources.

Good Programming Practice 8.6

Avoid reinventing the wheel. Study the capabilities of the Java API. If the API already contains a class that meets the requirements of your program, use that class rather than creating your own.

In general, to realize the full potential of software reusability, we need to improve cataloging schemes, licensing schemes, protection mechanisms that ensure master copies of classes are not corrupted, description schemes that system designers use to determine if existing objects meet their needs, browsing mechanisms that determine what classes are
available and how closely they meet software developer requirements, and the like. Many interesting research and development problems have been solved and many more need to be solved; these problems will be solved because the potential value of software reuse is enormous.

8.10 Final Instance Variables

We have emphasized repeatedly the principle of least privilege as one of the most fundamental principles of good software engineering. Let us see one way in which this principle applies to instance variables.

Some instance variables need to be modifiable and some do not. The programmer can use the keyword `final` to specify that a variable is not modifiable and that any attempt to modify the variable is an error. For example,

```java
private final int INCREMENT = 5;
```

declares a constant instance variable `INCREMENT` of type `int` and initializes it to 5.

Software Engineering Observation 8.17

Declaring an instance variable as `final` helps enforce the principle of least privilege. If an instance variable should not be modified, declare it to be `final` to expressly forbid modification.

Testing and Debugging Tip 8.3

Accidental attempts to modify a `final` instance variable are caught at compile time rather than causing execution-time errors. It is always preferable to get bugs out at compile time, if possible, rather than allowing them to slip through to execution time (where studies have found that the cost of repair is often as much as ten times more expensive).

Common Programming Error 8.9

Attempting to modify a `final` instance variable after it is initialized is a syntax error.

The applet of Fig. 8.11 creates a `final` instance variable `INCREMENT` of type `int` and initializes it to 5 in its declaration (line 15). A `final` variable cannot be modified by assignment after it is initialized. Such a variable can be initialized in its declaration or in every constructor of the class.

Common Programming Error 8.10

Not initializing a `final` instance variable in its declaration or in every constructor of the class is a syntax error.

```java
1 // Fig. 8.11: Increment.java
2 // Initializing a final variable
3 4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7
```

Fig. 8.11 Initializing a `final` variable (part 1 of 2).
Figure 8.12 illustrates compiler errors produced for the program of Fig. 8.11 if instance variable `INCREMENT` is declared `final`, but is not initialized in the declaration.

![Compiler error message as a result of not initializing increment.](image)

Figure 8.11 Initializing a `final` variable (part 2 of 2).

Figure 8.12 illustrates compiler errors produced for the program of Fig. 8.11 if instance variable `INCREMENT` is declared `final`, but is not initialized in the declaration.
8.11 Composition: Objects as Instance Variables of Other Classes

An `AlarmClock` class object needs to know when it is supposed to sound its alarm, so why not include a reference to a `Time` object as a member of the `AlarmClock` object? Such a capability is called *composition*. A class can have references to objects of other classes as members.

**Software Engineering Observation 8.18**

One form of software reuse is composition, in which a class has references to objects of other classes as members.

The next program contains three classes—`Date` (Fig. 8.13), `Employee` (Fig. 8.14) and `EmployeeTest` (Fig. 8.15). Class `Employee` contains instance variables `firstName`, `lastName`, `birthDate` and `hireDate`. Members `birthDate` and `hireDate` are references to `Date`s that contain instance variables `month`, `day` and `year`. This demonstrates that a class can contain references to objects of other classes. Class `EmployeeTest` instantiates an `Employee` and initializes and displays its instance variables. The `Employee` constructor (Fig. 8.14, lines 12–20) takes eight arguments—`first`, `last`, `birthMonth`, `birthDay`, `birthYear`, `hireMonth`, `hireDay` and `hireYear`. Arguments `birthMonth`, `birthDay` and `birthYear` are passed to the `Date` constructor (Fig. 8.13, lines 13–28) to initialize the `birthDate` object and `hireMonth`, `hireDay` and `hireYear` are passed to the `Date` constructor to initialize the `hireDate` object.

A member object does not need to be initialized immediately with constructor arguments. If an empty argument list is provided when a member object is created, the object’s default constructor (or no-argument constructor if one is available) will be called automatically. Values, if any, established by the default constructor (or no-argument constructor) can then be replaced by *set* methods.

**Performance Tip 8.2**

Initialize member objects explicitly at construction time by passing appropriate arguments to the constructors of the member objects. This eliminates the overhead of initializing member objects twice—once when the member object’s default constructor is called and again when *set* methods are used to provide initial values for the member object.

Note that both class `Date` (Fig. 8.13) and class `Employee` (Fig. 8.14) are defined as part of the package `com.deitel.jhtp4.ch08` as specified on line 3 of each file. Because they are in the same package (i.e., the same directory), class `Employee` does not need to import class `Date` to use it. When the compiler searches for the file `Date.class`, the compiler knows to search the directory where `Employee.class` is located. Classes in a package never need to import other classes from the same package.
private int day; // 1-31 based on month
private int year; // any year

// Constructor: Confirm proper value for month;
// call method checkDay to confirm proper
// value for day.
public Date( int theMonth, int theDay, int theYear )
{
    if ( theMonth > 0 && theMonth <= 12 ) // validate month
        month = theMonth;
    else {
        month = 1;
        System.out.println( "Month " + theMonth +
                " invalid. Set to month 1." );
    }

    year = theYear; // could validate year
day = checkDay( theDay ); // validate day

    System.out.println(
            "Date object constructor for date " + toString() );
}

// Utility method to confirm proper day value
// based on month and year.
private int checkDay( int testDay )
{
    int daysPerMonth[] =
        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    // check if day in range for month
    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
        return testDay;

    // check for leap year
    if ( month == 2 && testDay == 29 &&
            ( year % 400 == 0 ||
            ( year % 4 == 0 && year % 100 != 0 ) ) )
        return testDay;

    System.out.println( "Day " + testDay +
            " invalid. Set to day 1." );

    return 1; // leave object in consistent state
}

// Create a String of the form month/day/year
public String toString()
{
    return month + "/" + day + "/" + year;
}

} // end class Date

Fig. 8.13  Date class (part 2 of 2).
416

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Object-Based Programming

// Fig. 8.14: Employee.java
// Definition of class Employee.
package com.deitel.jhtp4.ch08;
public class Employee extends Object {
private String firstName;
private String lastName;
private Date birthDate;
private Date hireDate;
// constructor to initialize name, birth date and hire date
public Employee( String first, String last,
int birthMonth, int birthDay, int birthYear,
int hireMonth, int hireDay, int hireYear )
{
firstName = first;
lastName = last;
birthDate = new Date( birthMonth, birthDay, birthYear );
hireDate = new Date( hireMonth, hireDay, hireYear );
}
// convert Employee to String format
public String toString()
{
return lastName + ", " + firstName +
" Hired: " + hireDate.toString() +
" Birthday: " + birthDate.toString();
}
}

Fig. 8.14

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

Chapter 8

// end class Employee

Employee class with member object references.

// Fig. 8.15: EmployeeTest.java
// Demonstrating an object with a member object.
// Java extension packages
import javax.swing.JOptionPane;
// Deitel packages
import com.deitel.jhtp4.ch08.Employee;
public class EmployeeTest {

Fig. 8.15

// test class Employee
public static void main( String args[] )
{
Employee employee = new Employee( "Bob", "Jones",
7, 24, 49, 3, 12, 88 );

Demonstrating an object with a member object reference (part 1 of 2).
© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/3/01


8.12 Package Access

When no member access modifier is provided for a method or variable when it is defined in a class, the method or variable is considered to have package access. If your program consists of one class definition, this has no specific effects on the program. However, if your program uses multiple classes from the same package (i.e., a group of related classes), these classes can access each other’s package-access methods and data directly through a reference to an object.

Performance Tip 8.3

Package access enables objects of different classes to interact without the need for set and get methods that provide access to data, thus eliminating some of the method call overhead.

Let us consider a mechanical example of package access. The application of Fig. 8.16 contains two classes—the PackageDataTest application class (lines 8–34) and the PackageData class (lines 37–54). In the PackageData class definition, lines 38–39 declare the instance variables number and string with no member access modifiers; therefore, these are package access instance variables. The PackageDataTest application’s main method creates an instance of the PackageData class (line 13) to demonstrate the ability to modify the PackageData instance variables directly (as shown on lines 20–21). The results of the modification can be seen in the output window.

When you compile this program, the compiler produces two separate files—a .class file for class PackageData and a .class file for class PackageDataTest. Every Java class has its own .class file. These two .class files are placed in the same directory by the compiler automatically and are considered to be part of the same package (they are certainly related by the fact that they are in the same file). Because they are part of the same package, class PackageDataTest is allowed to modify the package access data of objects of class PackageData.
// Fig. 8.16: PackageDataTest.java
// Classes in the same package (i.e., the same directory) can
// use package access data of other classes in the same package.

import javax.swing.JOptionPane;

public class PackageDataTest {
    public static void main( String args[] )
    {
        PackageData packageData = new PackageData();

        // append String representation of packageData to output
        String output =
            "After instantiation:
" + packageData.toString();

        // change package access data in packageData object
        packageData.number = 77;
        packageData.string = "Good bye";

        // append String representation of packageData to output
        output += 
            "After changing values:
" + packageData.toString();

        JOptionPane.showMessageDialog( null, output,
            "Demonstrating Package Access",
            JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }
}

// class with package access instance variables
class PackageData {
    int number;     // package access instance variable
    String string;  // package access instance variable

    // constructor
    public PackageData()
    {
        number = 0;
        string = "Hello";
    }

    // convert PackageData object to String representation
    public String toString()
    {
        return "number: " + number + " string: " + string;
    }
}
Software Engineering Observation 8.19

Some people in the OOP community feel that package access corrupts information hiding and weakens the value of the object-oriented design approach, because the programmer must assume responsibility for error checking and data validation in any code that manipulates the package access data members.

8.13 Using the this Reference

When a method of a class references another member of that class for a specific object of that class, how does Java ensure that the proper object is referenced? The answer is that each object has access to a reference to itself—called the this reference.

The this reference is used implicitly to refer to both the instance variables and methods of an object. We begin with a simple example of using the this reference explicitly; a subsequent example shows some substantial and subtle examples of using this.

Performance Tip 8.4

Java conserves storage by maintaining only one copy of each method per class; this method is invoked by every object of that class. Each object, on the other hand, has its own copy of the class’s instance variables.

The application of Fig. 8.17 demonstrates implicit and explicit use of the this reference to enable the main method of class ThisTest to display the private data of a SimpleTime object.
public class ThisTest {
    // test class SimpleTime
    public static void main( String args[] )
    {
        SimpleTime time = new SimpleTime( 12, 30, 19 );
        JOptionPane.showMessageDialog( null, time.buildString(),
            "Demonstrating the \"this\" Reference",
            JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 );
    }
} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime {
    private int hour, minute, second;

    // constructor uses parameter names identical to instance
    // variable names, so "this" reference required to distinguish
    // between instance variables and parameters
    public SimpleTime( int hour, int minute, int second )
    {
        this.hour = hour;  // set "this" object's hour
        this.minute = minute; // set "this" object's minute
        this.second = second; // set "this" object's second
    }

    // call toString explicitly via "this" reference, explicitly
    // via implicit "this" reference, implicitly via "this"
    public String buildString()
    {
        return "this.toString(): " + this.toString() +
            "toString(): " + toString() +
            "this (with implicit toString() call): " + this;
    }

    // convert SimpleTime to String format
    public String toString()
    {
        DecimalFormat twoDigits = new DecimalFormat( "00" );
        return twoDigits.format( this.hour ) + ":" +
            twoDigits.format( this.minute ) + ":" +
            twoDigits.format( this.second );
    }
} // end class SimpleTime

Fig. 8.17 Using the this reference implicitly and explicitly (part 2 of 3).
Class `SimpleTime` (lines 8–60) defines three `private` instance variables—`hour`, `minute` and `second`. The constructor (lines 34–39) receives three `int` arguments to initialize a `SimpleTime` object. Notice that the parameter names for the constructor are the same as the instance variable names. Remember that a local variable of a method with the same name as an instance variable of a class hides the instance variable in the scope of the method. For this reason, we use the `this` reference to refer explicitly to the instance variables on lines 36–38.

**Common Programming Error 8.11**

In a method in which a method parameter has the same name as one of the class members, use `this` explicitly if you want to access the class member; otherwise, you will incorrectly reference the method parameter.

**Good Programming Practice 8.7**

Avoid using method parameter names that conflict with class member names.

Method `buildString` (lines 43–48) returns a `String` created with a statement that uses the `this` reference three ways. Line 45 explicitly invokes the class’s `toString` method via `this.toString()`. Line 46 implicitly uses the `this` reference to perform the same task. Line 47 appends `this` to the string that will be returned. Remember that the `this` reference is a reference to an object—the current `SimpleTime` object being manipulated. As before, any reference added to a `String` results in a call to the `toString` method for the referenced object. Method `buildString` is invoked at line 18 to display the results of the three calls to `toString`. Note that the same time is displayed on all three lines of the output because all three calls to `toString` are for the same object.

Another use of the `this` reference is in enabling concatenated method calls (also called *cascaded method calls* or *method call chaining*). Figure 8.18 illustrates returning a reference to a `Time4` object to enable method calls of class `Time4` to be concatenated. Methods `setTime` (lines 50–57), `setHour` (lines 60–65), `setMinute` (line 68–74) and `setSecond` (lines 77–83) each have a return type of `Time4` and each has as its last statement

```java
    return this;
```

to indicate that a reference to the `Time4` object being manipulated should be returned to the caller of the method.
// Fig. 8.18: Time4.java
// Time4 class definition
package com.deitel.jhtp4.ch08;

// Java core packages
import java.text.DecimalFormat;

public class Time4 extends Object {

private int hour; // 0 - 23
private int minute; // 0 - 59
private int second; // 0 - 59

// Time4 constructor initializes each instance variable
to zero. Ensures that Time object starts in a consistent state.
public Time4()
{
    this.setTime( 0, 0, 0 );
}

// Time4 constructor: hour supplied, minute and second defaulted to 0
public Time4( int hour )
{
    this.setTime( hour, 0, 0 );
}

// Time4 constructor: hour and minute supplied, second defaulted to 0
public Time4( int hour, int minute )
{
    this.setTime( hour, minute, 0 );
}

// Time4 constructor: hour, minute and second supplied
public Time4( int hour, int minute, int second )
{
    this.setTime( hour, minute, second );
}

// Time4 constructor: another Time4 object supplied.
public Time4( Time4 time )
{
    this.setTime( time.getHour(), time.getMinute(),
                 time.getSecond() );
}

// Set Methods
// set a new Time value using universal time
public Time4 setTime( int hour, int minute, int second )
{
    this.setHour( hour ); // set hour
```java
53     this.setMinute( minute ); // set minute
54     this.setSecond( second ); // set second
55     
56     return this;    // enables chaining
57 }
58
59 // validate and set hour
60 public Time4 setHour( int hour )
61 {
62     this.hour = ( hour >= 0 && hour < 24 ? hour : 0 );
63     
64     return this;    // enables chaining
65 }
66
67 // validate and set minute
68 public Time4 setMinute( int minute )
69 {
70     this.minute =
71     ( minute >= 0 && minute < 60 ) ? minute : 0;
72     
73     return this;    // enables chaining
74 }
75
76 // validate and set second
77 public Time4 setSecond( int second )
78 {
79     this.second =
80     ( second >= 0 && second < 60 ) ? second : 0;
81     
82     return this;    // enables chaining
83 }
84
85 // Get Methods
86 // get value of hour
87 public int getHour() { return this.hour; }
88
89 // get value of minute
90 public int getMinute() { return this.minute; }
91
92 // get value of second
93 public int getSecond() { return this.second; }
94
95 // convert to String in universal-time format
96 public String toUniversalString()
97 {
98     DecimalFormat twoDigits = new DecimalFormat( "00" );
99     
100    return twoDigits.format( this.getHour() ) + ":" +
101     twoDigits.format( this.getMinute() ) + ":" +
102     twoDigits.format( this.getSecond() );
103 }
104```

Fig. 8.18  Class Time4 using this to enable chained method calls (part 2 of 3).
The example again demonstrates the explicit use of the `this` reference inside the body of a class. In class `Time4`, every use of an instance variable of the class and every call to another method in class `Time4` uses the `this` reference explicitly. Most programmers prefer not to use the `this` reference unless it is required or helps clarify a piece of code.

**Good Programming Practice 8.8**

Explicitly using `this` can increase program clarity in some contexts in which `this` is optional.

In application class `TimeTest6` (Fig. 8.19), line 18 and line 26 both demonstrate method call chaining. Why does the technique of returning the `this` reference work? Let us discuss line 18. The dot operator (.) associates from left to right, so the expression

\[
\text{time.setHour(18).setMinute(30).setSecond(22);}
\]

first evaluates `time.setHour(18)`, then returns a reference to object `time` as the result of this method call. Any time you have a reference in a program (even as the result of a method call), the reference can be followed by a dot operator and a call to one of the methods of the reference type. Thus, the remaining expression is interpreted as

\[
\text{time.setMinute(30).setSecond(22);}
\]

The `time.setMinute(30)` call executes and returns a reference to `time`. The remaining expression is interpreted as

\[
\text{time.setSecond(22);}
\]

When the statement is complete, the time is 18 for the `hour`, 30 for the `minute` and 22 from the `second`.

Note that the calls on line 26

\[
\text{time.setTime(20,20,20).toString();}
\]

also use the concatenation feature. These method calls must appear in this order in this expression because `toString` as defined in the class does not return a reference to a `Time4` object. Placing the call to `toString` before the call to `setTime` causes a syntax error. Note that `toString` returns a reference to a `String` object. Therefore, a method of class `String` could be concatenated to the end of line 26.
Note that the purpose of the example in Fig. 8.18 and Fig. 8.19 is to demonstrate the mechanics of concatenated method calls. Many Java methods return references to objects that can be used in the manner shown here. It is important to understand concatenated method calls as they appear frequently in Java programs.

**Good Programming Practice 8.9**

For program clarity, avoid using concatenated method calls.
8.14 Finalizers

We have seen that constructor methods are capable of initializing data in an object of a class when the class is created. Often, constructors acquire various system resources such as memory (when the `new` operator is used). We need a disciplined way to give resources back to the system when they are no longer needed to avoid resource leaks. The most common resource acquired by constructors is memory. Java performs automatic garbage collection of memory to help return memory back to the system. When an object is no longer used in the program (i.e., there are no references to the object), the object is marked for garbage collection. The memory for such an object can be reclaimed when the garbage collector executes. Therefore, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely to happen in Java. However, other resource leaks can occur.

**Performance Tip 8.5**

Extensive use of local variables that refer to objects can degrade performance. When a local variable is the only reference to an object, the object is marked for garbage collection as the local variable goes out of scope. If this happens frequently in short time periods, large numbers of objects could be marked for garbage collection, thus placing a performance burden on the garbage collector.

Every class in Java can have a finalizer method that returns resources to the system. The finalizer method for an object is guaranteed to be called to perform termination housekeeping on the object just before the garbage collector reclaims the memory for the object. A class’s finalizer method always has the name `finalize`, receives no parameters and returns no value (i.e., its return type is `void`). A class should have only one `finalize` method that takes no arguments. Method `finalize` is defined originally in class `Object` as a placeholder that does nothing. This guarantees that every class has a `finalize` method for the garbage collector to call.

**Good Programming Practice 8.10**

The last statement in a `finalize` method should always be `super.finalize();` to ensure that the superclass’s `finalize` method is called.

**Software Engineering Observation 8.20**

The garbage collector is not guaranteed to execute; therefore, an object’s `finalize` method is not guaranteed to get called. You should not architect classes that rely on the garbage collector calling an object’s `finalize` method to deallocate resources.

Finalizers have not been provided for the classes presented so far. Actually, finalizers are rarely used in industrial Java applications. We will see a sample `finalize` method and discuss the garbage collector further in Figure 8.20.

**Testing and Debugging Tip 8.4**

Several professional Java developers who reviewed this book indicated that method `finalize` is not useful in industrial Java applications, because it is not guaranteed to get called. For this reason, you should search your Java programs for any use of method `finalize` to ensure that the program does not rely on calls to method `finalize` for proper deallocation of resources. In fact, you might consider removing method `finalize` entirely and
8.15 Static Class Members

Each object of a class has its own copy of all the instance variables of the class. In certain cases, only one copy of a particular variable should be shared by all objects of a class. A static class variable is used for these and other reasons. A static class variable represents class-wide information—all objects of the class share the same piece of data. The declaration of a static member begins with the keyword static.

Let us motivate the need for static class-wide data with a video game example. Suppose we have a video game with Martians and other space creatures. Each Martian tends to be brave and willing to attack other space creatures when the Martian is aware that there are at least five Martians present. If there are fewer than five Martians present, each Martian becomes cowardly. So each Martian needs to know the martianCount. We could endow class Martian with martianCount as instance data. If we do this, then every Martian will have a separate copy of the instance data and every time we create a new Martian we will have to update the instance variable martianCount in every Martian. This wastes space with the redundant copies and wastes time in updating the separate copies. Instead, we declare martianCount to be static. This makes martianCount class-wide data. Every Martian can see the martianCount as if it were instance data of the Martian, but only one copy of the static martianCount is maintained by Java. This saves space. We save time by having the Martian constructor increment the static martianCount. Because there is only one copy, we do not have to increment separate copies of martianCount for each Martian object.

Performance Tip 8.6

Use static class variables to save storage when a single copy of the data will suffice.

Although static class variables may seem like global variables, static class variables have class scope. A class’s public static class members can be accessed through a reference to any object of that class, or they can be accessed through the class name by using the dot operator (e.g., Math.random()). A class’s private static class members can be accessed only through methods of the class. Actually, static class members exist even when no objects of that class exist—they are available as soon as the class is loaded into memory at execution time. To access a public static class member when no objects of the class exist, simply prefix the class name and the dot operator to the class member. To access a private static class member when no objects of the class exist, a public static method must be provided and the method must be called by prefixing its name with the class name and dot operator.

Our next program defines two classes—Employee (Fig. 8.20) and EmployeeTest (Fig. 8.21). Class Employee defines a private static class variable and a public static method. The class variable count (Fig. 8.20, line 6) is initialized to zero by default. Class variable count maintains a count of the number of objects of class Employee that have been instantiated and currently reside in memory. This includes objects that have already been marked for garbage collection but have not yet been reclaimed.
When objects of class `Employee` exist, member `count` can be used in any method of an `Employee` object—in this example, `count` is incremented (line 15) by the con-
structor and decremented (line 25) by the finalizer. When no objects of class Employee exist, member count can still be referenced, but only through a call to public static method getCount (lines 43–46), as in:

Employee.getCount()

which determines the number of Employee objects currently in memory. Note that when there are no objects instantiated in the program, the Employee.getCount() method call is issued. However, when there are objects instantiated, method getCount can also be called through a reference to one of the objects, as in

e1.getCount()

Good Programming Practice 8.11

Always invoke static methods by using the class name and the dot operator (.). This emphasizes to other programmers reading your code that the method being called is a static method.

Notice that the Employee class has a finalize method (lines 23–28). This method is included to show when it is called by the garbage collector in a program. Method finalize normally is declared protected so it is not part of the public services of a class. We will discuss the protected access modifier in detail in Chapter 9.

Method main of the EmployeeTest application class (Fig. 8.21) instantiates two Employee objects (lines 16–17). When each Employee object’s constructor is invoked, lines 12–13 of Fig. 8.20 store references to that Employee’s first name and last name String objects. Note that these two statements do not make copies of the original Strings arguments. Actually, String objects in Java are immutable—they cannot be modified after they are created (class String does not provide any set methods). Because a reference cannot be used to modify a String, it is safe to have many references to one String object in a Java program. This is not normally the case for most other classes in Java. If Java String objects are immutable, why are we able to use the + and += operators to concatenate Strings? As we discuss in Chapter 10, Strings and Characters, the String concatenation operations actually result in the creation of new String objects containing the concatenated values. The original String objects actually are not modified.

When main is done with the two Employee objects, the references e1 and e2 are set to null at lines 37–38. At this point references e1 and e2 no longer refer to the objects that were instantiated on lines 16–17. This marks the objects for garbage collection because there are no more references to the objects in the program.

Fig. 8.21 Using a static class variable to maintain a count of the number of objects of a class (part 1 of 3).
public class EmployeeTest {

    // test class Employee
    public static void main( String args[] )
    {
        // prove that count is 0 before creating Employees
        String output = "Employees before instantiation: " +
                        Employee.getCount();

        // create two Employees; count should be 2
        Employee e1 = new Employee( "Susan", "Baker" );
        Employee e2 = new Employee( "Bob", "Jones" );

        // Prove that count is 2 after creating two Employees.
        // Note: static methods should be called only via the
        // class name for the class in which they are defined.
        output += "

Employees after instantiation: " +
        "\nvia e1.getCount(): " + e1.getCount() +
        "\nvia e2.getCount(): " + e2.getCount() +
        "\nvia Employee.getCount(): " + Employee.getCount();

        // get names of Employees
        output += "

Employee 1: " + e1.getFirstName() +
        " " + e1.getLastName() +
        "Employee 2: " +
        e2.getFirstName() + " " + e2.getLastName();

        // If there is only one reference to each employee (as
        // on this example), the following statements mark
        // those objects for garbage collection. Otherwise,
        // these statement simply decrement the reference count
        // for each object.
        e1 = null;
        e2 = null;

        System.gc(); // suggest call to garbage collector

        // Show Employee count after calling garbage collector.
        // Count displayed may be 0, 1 or 2 depending on
        // whether garbage collector executed immediately and
        // number of Employee objects it collects.
        output += "\n\nEmployees after System.gc(): " +
        Employee.getCount();

        JOptionPane.showMessageDialog( null, output,
        "Static Members and Garbage Collection",
        JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }

} // end class EmployeeTest

Fig. 8.21 Using a static class variable to maintain a count of the number of
objects of a class (part 2 of 3).
Eventually, the garbage collector reclaims the memory for these objects (or the memory is reclaimed by the operating system when the program terminates). Because it is not guaranteed when the garbage collector will execute, we make an explicit call to the garbage collector with line 40, which uses `public static` method `gc` from class `System` (`java.lang` package) to indicate that the garbage collector immediately should make a best effort attempt to collect objects that have been marked for garbage collection. However, this is just a best effort—it is possible that no objects or a subset of the garbage objects will be collected. In our example, the garbage collector did execute before lines 49–51 displayed the results of the program. The last line of the output indicates that the number of `Employee` objects in memory is 0 after the call to `System.gc()`. Also, the last two lines of the command window output show that the `Employee` object for `Susan Baker` was finalized before the `Employee` object for `Bob Jones`. Remember, the garbage collector is not guaranteed to execute when `System.gc()` is invoked nor is it guaranteed to collect objects in a specific order, so it is possible that the output of this program on your system may differ.

*Note: A method declared `static` cannot access non-`static` class members. Unlike non-`static` methods, a `static` method has no `this` reference because, `static` class methods exist independent of any objects of a class and before any objects of the class have been instantiated.*

**Common Programming Error 8.12**

Referring to the `this` reference in a `static` method is a syntax error.

**Common Programming Error 8.13**

It is a syntax error for a `static` method to call an instance method or to access an instance variable.
Software Engineering Observation 8.21

Any **static** class variables and **static** class methods exist and can be used even if no objects of that class have been instantiated.

### 8.16 Data Abstraction and Encapsulation

Classes normally hide their implementation details from the clients of the classes. This is called *encapsulation* or *information hiding*. As an example of encapsulation, let us consider a data structure called a **stack**.

Think of a stack in terms of a pile of dishes. When a dish is placed on the pile, it is always placed at the top (referred to as **pushing** the dish onto the stack), and when a dish is removed from the pile, it is always removed from the top (referred to as **popping** the dish off the stack). Stacks are known as *last-in, first-out (LIFO) data structures*—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

The programmer can create a stack class and hide from its clients implementation of the stack. Stacks can easily be implemented with arrays and other methods (such as linked lists; see Chapter 19, “Data Structures,” and Chapter 20, “Java Utilities Packages and Bit Manipulation”). A client of a stack class need not know how the stack is implemented. The client simply requires that when data items are placed in the stack, they will be recalled in last-in, first-out order. This concept is referred to as **data abstraction**, and Java classes define abstract data types (ADTs). Although users might happen to know the details of how a class is implemented, users may not write code that depends on these details. This means that a particular class (such as one that implements a stack and its operations of **push** and **pop**) can be replaced with another version without affecting the rest of the system, as long as the public services of that class does not change (i.e., every method still has the same name, return type and parameter list in the new class definition).

The job of a high-level language is to create a view convenient for programmers to use. There is no single accepted standard view—that is one reason why there are so many programming languages. Object-oriented programming in Java presents yet another view.

Most programming languages emphasize actions. In these languages, data exists in support of the actions programs need to take. Data is “less interesting” than actions, anyway. Data is “crude.” There are only a few built-in data types, and it is difficult for programmers to create their own new data types.

This view changes with Java and the object-oriented style of programming. Java elevates the importance of data. The primary activity in Java is creating new data types (i.e., classes) and expressing the interactions among objects of those data types.

To move in this direction, the programming-languages community needed to formalize some notions about data. The formalization we consider is the notion of *abstract data types (ADTs)*. ADTs receive as much attention today as structured programming did over the last two decades. ADTs do not replace structured programming. Rather, they provide an additional formalization to further improve the program development process.

What is an abstract data type? Consider the built-in type **int**. What comes to mind is the notion of an integer in mathematics, but **int** on a computer is not precisely what an integer is in mathematics. In particular, computer **ints** are normally quite limited in size. For example, **int** on a 32-bit machine is limited approximately to the range –2 billion to +2 billion. If the result of a calculation falls outside this range, an error occurs and the
machine responds in some machine-dependent manner, including the possibility of “quietly” producing an incorrect result. Mathematical integers do not have this problem. So the notion of a computer int is really only an approximation to the notion of a real-world integer. The same is true with float.

The point is that even the built-in data types provided with programming languages like Java are really only approximations or models of real-world concepts and behaviors. We have taken int for granted until this point, but now you have a new perspective to consider. Types like int, float, char and others are all examples of abstract data types. They are essentially ways of representing real-world notions to some satisfactory level of precision within a computer system.

An abstract data type actually captures two notions, namely a data representation and the operations that are allowed on that data. For example, the notion of int defines addition, subtraction, multiplication, division and modulus operations in Java, but division by zero is undefined. Another example is the notion of negative integers whose operations and data representation are clear, but the operation of taking the square root of a negative integer is undefined. In Java, the programmer uses classes to implement abstract data types.

Java has a small set of primitive types. ADTs extend the base programming language. Software Engineering Observation 8.22

The programmer is able to create new types through the use of the class mechanism. These new types can be designed to be used as conveniently as the built-in types. Thus, Java is an extensible language. Although the language is easy to extend with these new types, the base language itself is not changeable.

New Java classes can be proprietary to an individual, to small groups, to companies, and so on. Many classes are placed in standard class libraries intended for wide distribution. This does not necessarily promote standards, although de facto standards are emerging. The full value of Java will be realized only when substantial, standardized class libraries become more widely available than they are today. In the United States, such standardization often happens through ANSI, the American National Standards Institute. Worldwide standardization often happens through ISO, the International Standards Organization. Regardless of how these libraries ultimately appear, the reader who learns Java and object-oriented programming will be ready to take advantage of the new kinds of rapid, component-oriented software development made possible with class libraries.

8.16.1 Example: Queue Abstract Data Type

Each of us stands in line from time to time. A waiting line is also called a queue. We wait in line at the supermarket checkout counter, we wait in line to get gasoline, we wait in line to board a bus, we wait in line to pay a toll on the highway, and students know all too well about waiting in line during registration to get the courses they want. Computer systems use many waiting lines internally, so we write programs that simulate what queues are and do.

A queue is a good example of an abstract data type. A queue offers well-understood behavior to its clients. Clients put things in a queue one at a time—using an enqueue operation, and the clients get those things back one at a time on demand—using a dequeue operation. Conceptually, a queue can become infinitely long. A real queue, of course, is finite. Items are returned from a queue in first-in, first-out (FIFO) order—the first item inserted in the queue is the first item removed from the queue.
The queue hides an internal data representation that keeps track of the items currently waiting in line, and it offers a set of operations to its clients, namely enqueue and dequeue. The clients are not concerned about implementation of the queue. Clients merely want the queue to operate “as advertised.” When a client enqueues a new item, the queue should accept that item and place it internally in some kind of first-in, first-out data structure. When the client wants the next item from the front of the queue, the queue should remove the item from its internal representation and should deliver the item to the outside world in FIFO order (i.e., the item that has been in the queue the longest should be the next one returned by the next dequeue operation).

The queue ADT guarantees the integrity of its internal data structure. Clients must not manipulate this data structure directly. Only the queue ADT has access to its internal data (i.e., the queue ADT encapsulates its data). Clients must cause only allowable operations to be performed on the data representation; operations not provided in the ADT’s public interface are rejected by the ADT in some appropriate manner. This could mean issuing an error message, terminating execution, or simply ignoring the operation request.

8.17 (Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulation

In the “Thinking About Objects” sections in Chapters 1 through 7, we introduced the fundamentals of object orientation and developed an object-oriented design for our elevator simulation. In Chapter 8, we introduced the details of programming with Java classes. We now begin implementing our object-oriented design in Java. At the end of this section, we show how to generate code in Java, working from class diagrams. This process is referred to as forward engineering.1

Visibility
Before we begin implementing our design in Java, we apply member-access modifiers (see Section 8.2) to the members of our classes. In Chapter 8, we introduced the access specifiers public and private—these determine the visibilities of an object’s attributes and methods to other objects. Before we create class files, we consider which attributes and methods of our classes should be public and which should be private.

Software Engineering Observation 8.23

Each element of a class should have private visibility unless it can be proven that the element needs public visibility.

In Chapter 8, we discussed how attributes generally should be private, but what about the operations of a class—its methods? These operations are invoked by clients of that class; therefore, the methods normally should be public. In the UML, public visibility is indicated by placing a plus sign (+) before a particular element (i.e., a method or an attribute); a minus sign (-) indicates private visibility. Figure 8.13 shows our updated class diagram with visibility notations included.

Implementation: Forward Engineering

Forward engineering is the process of transforming a design, such as that in a class diagram, into code of a specific programming language, such as Java. Now that we have discussed programming Java classes, we forward engineer the class diagram of Fig. 8.13 into the Java code for our elevator simulator. The generated code will represent only the “skeleton,” or the structure, of the model.\(^2\) In Chapters 9 and 10, we modify the code to incorporate inheritance and interfaces, respectively. In Appendix G, Appendix H and Appendix I, we present the complete, working Java code for our model.

As an example, we forward engineer class \texttt{Elevator} from Fig. 8.13. We use this figure to determine the attributes and operations of that class. We use the class diagram of Fig. 3.23 to determine associations (and aggregations) among classes. We adhere to the following four guidelines:

1. Use the name located in the first compartment to declare the class as a \texttt{public} class with an empty constructor. For example, class \texttt{Elevator} yields

2. So far, we have presented only about half of the case-study material—we have not yet discussed inheritance, event handling, multithreading and animation. The standard development process recommends finishing the design process before starting the coding process. Technically, we will not have finished designing our system until we have discussed these additional topics, so our current code implementation might seem premature. We present only a partial implementation illustrating the topics covered in Chapter 8.

\(^2\) In Chapters 9 and 10, we modify the code to incorporate inheritance and interfaces, respectively. In Appendix G, Appendix H and Appendix I, we present the complete, working Java code for our model.
public class Elevator {

    public Elevator() {} 
}

2. Use the attributes located in the second compartment to declare the member variables. For example, the private attributes moving, summoned, currentFloor, destinationFloor, capacity and travelTime of class Elevator yield

    public class Elevator {

        // class attributes
        private boolean moving;
        private boolean summoned;
        private int currentFloor = 1;
        private int destinationFloor = 2;
        private int capacity = 1;
        private int travelTime = 5;

        // class constructor
        public Elevator() {} 
    }

3. Use the associations described in the class diagram to generate the references to other objects. For example, according to Fig. 3.23, Elevator contains one object each of classes ElevatorDoor, ElevatorButton and Bell. This yields

    public class Elevator {

        // class attributes
        private boolean moving;
        private boolean summoned;
        private int currentFloor = 1;
        private int destinationFloor = 2;
        private int capacity = 1;
        private int travelTime = 5;

        // class objects
        private ElevatorDoor elevatorDoor;
        private ElevatorButton elevatorButton;
        private Bell bell;

        // class constructor
        public Elevator() {} 
    }

4. Use the operations located in the third compartment of Fig. 8.13 to declare the methods. For example, the public operations ride, requestElevator, enterElevator, exitElevator and departElevator in Elevator yield
public class Elevator {

    // class attributes
    private boolean moving;
    private boolean summoned;
    private int currentFloor = 1;
    private int destinationFloor = 2;
    private int capacity = 1;
    private int travelTime = 5;

    // class objects
    private ElevatorDoor elevatorDoor;
    private ElevatorButton elevatorButton;
    private Bell bell;

    // class constructor
    public Elevator() {}

    // class methods
    public void ride() {}
    public void requestElevator() {}
    public void enterElevator() {}
    public void exitElevator() {}
    public void departElevator() {}
}

This concludes the basics of forward engineering. We return to this example at the ends of “Thinking About Objects” Section 9.23 and Section 10.22 to incorporate inheritance, interfaces and event handling.

SUMMARY

• OOP encapsulates data (attributes) and methods (behaviors) into objects; the data and methods of an object are intimately tied together.

• Objects have the property of information hiding. Objects might know how to communicate with one another across well-defined interfaces, but they normally are not allowed to know how other objects are implemented.

• Java programmers concentrate on creating their own user-defined types called classes.

• The non-static data components of a class are called instance variables. The static data components of a class are called class variables.

• Java uses inheritance to create new classes from existing class definitions.

• Every class in Java is a subclass of Object. Thus, every new class definition has the attributes (data) and behaviors (methods) of class Object.

• Keywords public and private are member access modifiers.

• Instance variables and methods declared with member access modifier public are accessible wherever the program has a reference to an object of the class in which they are defined.

• Instance variables and methods declared with member access modifier private are accessible only to methods of the class in which they are defined.

• Instance variables are normally declared private and methods are normally declared public.

• The public methods (or public services) of a class are used by clients of the class to manipulate the data stored in objects of the class.
A constructor is a method with the exact same name as the class that initializes the instance variables of an object of the class when the object is instantiated. Constructor methods can be overloaded for a class. Constructors can take arguments but cannot specify a return value type.

Constructors and other methods that change instance variable values should always maintain objects in a consistent state.

Method `toString` takes no arguments and returns a `String`. The original `toString` method of class `Object` is a placeholder that is normally redefined by a subclass.

When an object is instantiated, operator `new` allocates the memory for the object, then `new` calls the constructor for the class to initialize the instance variables of the object.

If the `.class` files for the classes used in a program are in the same directory as the class that uses them, `import` statements are not required.

Concatenating a `String` and any object results in an implicit call to the object’s `toString` method to convert the object to a `String`, then the `Strings` are concatenated.

Within a class’s scope, class members are accessible to all of that class’s methods and can be referenced simply by name. Outside a class’s scope, class members can only be accessed off a “handle” (i.e., a reference to an object of the class).

If a method defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the method-scope variable in the method. A hidden instance variable can be accessed in the method by preceding its name with the keyword `this` and the dot operator.

Each class and interface in the Java API belongs to a specific package that contains a group of related classes and interfaces.

Packages are actually directory structures used to organize classes and interfaces. Packages provide a mechanism for software reuse and a convention for unique class names.

Creating a reusable class requires: defining a `public` class, adding a `package` statement to the class definition file, compiling the class into the appropriate package directory structure and importing the class into a program.

When compiling a class in a package, the option `-d` must be passed to the compiler to specify where to create (or locate) all the directories in the `package` statement.

The `package` directory names become part of the class name when the class is compiled. Use this fully qualified name in programs or `import` the class and use its short name (the name of the class by itself) in the program.

If no constructors are defined for a class, the compiler creates a default constructor.

When one object of a class has a reference to another object of the same class, the first object can access all the second object’s data and methods.

Classes often provide `public` methods to allow clients of the class to `set` (i.e., assign values to) or `get` (i.e., obtain the values of) `private` instance variables. `Get` methods are also commonly called accessor methods or query methods. `Set` methods are also commonly called mutator methods (because they typically change a value).

Every event has a source—the GUI component with which the user interacted to signal the program to do a task.

Use the keyword `final` to specify that a variable is not modifiable and that any attempt to modify the variable is an error. A `final` variable cannot be modified by assignment after it is initialized. Such a variable must be initialized in its declaration or in every constructor of the class.

With composition, a class has references to objects of other classes as members.

When no member access modifier is provided for a method or variable when it is defined in a class, the method or variable is considered to have package access.
• If a program uses multiple classes from the same package, these classes can access each other’s package-access methods and data directly through a reference to an object.

• Each object has access to a reference to itself called the this reference that can be used inside the methods of the class to refer to the object’s data and other methods explicitly.

• Any time you have a reference in a program (even as the result of a method call), the reference can be followed by a dot operator and a call to one of the methods for the reference type.

• Java performs automatic garbage collection of memory. When an object is no longer used in the program (i.e., there are no references to the object), the object is marked for garbage collection.

• Every class in Java can have a finalizer method that returns resources to the system. A class’s finalizer method always has the name finalize, receives no parameters and returns no value. Method finalize is originally defined in class Object as a placeholder that does nothing. This guarantees that every class has a finalize method for the garbage collector to call.

• A static class variable represents class-wide information—all objects of the class share the same piece of data. A class’s public static members can be accessed through a reference to any object of that class, or they can be accessed through the class name using the dot operator.

• public static method gc from class System suggests that the garbage collector immediately make a best effort attempt to collect garbage objects. The garbage collector is not guaranteed to collect objects in a specific order.

• A method declared static cannot access nonstatic class members. Unlike nonstatic methods, a static method has no this reference, because static class variables and static class methods exist independent of any objects of a class.

• static class members exist even when no objects of that class exist—they are available as soon as the class is loaded into memory at execution time.

TERMINOLOGY

abstract data type (ADT) extends
access method extensibility
aggregation finalize
attribute get method
behavior helper method
cascaded method calls implementation of a class
class information hiding
class definition initialize a class object
class library instance method
class method (static) instance of a class
class scope instance variable
class variable instantiate an object of a class
client of a class interface to a class
composition member access control
concatenated method calls member access modifiers
consistent state for an instance variable
constructor member access operator (.)
data type new operator
default constructor no-argument constructor
dot operator (.) object
SELF-REVIEW EXERCISES

8.1 Fill in the blanks in each of the following statements:
   a) Class members are accessed via the ______ operator in conjunction with a reference to an object of the class.
   b) Members of a class specified as ______ are accessible only to methods of the class.
   c) A ______ is a special method used to initialize the instance variables of a class.
   d) A ______ method is used to assign values to ______ instance variables of a class.
   e) Methods of a class are normally made ______ and instance variables of a class are normally made ______.
   f) A ______ method is used to retrieve values of ______ data of a class.
   g) The keyword ______ introduces a class definition.
   h) Members of a class specified as ______ are accessible anywhere an object of the class is in scope.
   i) The ______ operator dynamically allocates memory for an object of a specified type and returns a ______ to that type.
   j) A ______ instance variable represents class-wide information.
   k) The keyword ______ specifies that an object or variable is not modifiable after it is initialized.
   l) A method declared ______ cannot access ______ class members.

ANSWERS TO SELF-REVIEW EXERCISES

8.1  a) dot (.), b) ______. c) constructor. d) set. e) ______, ______. f) get. g) ______.
     h) ______. i) ______, ______. j) ______. k) ______. l) ______.

EXERCISES

8.2 Create a class called ______ for performing arithmetic with complex numbers. Write a driver program to test your class.

Complex numbers have the form

\[ \text{realPart} + \text{imaginaryPart} \times i \]

where \( i \) is

\[ \sqrt{-1} \]

Use floating-point variables to represent the ______ data of the class. Provide a constructor method that enables an object of this class to be initialized when it is declared. Provide a no-argu-
ment constructor with default values in case no initializers are provided. Provide public methods for each of the following:
   a) Addition of two Complex numbers: The real parts are added together and the imaginary parts are added together.
   b) Subtraction of two Complex numbers: The real part of the right operand is subtracted from the real part of the left operand and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.
   c) Printing Complex numbers in the form \((a, b)\), where \(a\) is the real part and \(b\) is the imaginary part.

8.3 Create a class called Rational for performing arithmetic with fractions. Write a driver program to test your class.

   Use integer variables to represent the private instance variables of the class—the numerator and the denominator. Provide a constructor method that enables an object of this class to be initialized when it is declared. The constructor should store the fraction in reduced form (i.e., the fraction \(2/4\) would be stored in the object as 1 in the numerator and 2 in the denominator). Provide a no-argument constructor with default values in case no initializers are provided. Provide public methods for each of the following:
   a) Addition of two Rational numbers. The result of the addition should be stored in reduced form.
   b) Subtraction of two Rational numbers. The result of the subtraction should be stored in reduced form.
   c) Multiplication of two Rational numbers. The result of the multiplication should be stored in reduced form.
   d) Division of two Rational numbers. The result of the division should be stored in reduced form.
   e) Printing Rational numbers in the form \(a/b\), where \(a\) is the numerator and \(b\) is the denominator.
   f) Printing Rational numbers in floating-point format. (Consider providing formatting capabilities that enable the user of the class to specify the number of digits of precision to the right of the decimal point.)

8.4 Modify the Time3 class of Fig. 8.8 to include the tick method that increments the time stored in a Time3 object by one second. Also provide method incrementMinute to increment the minute and method incrementHour to increment the hour. The Time3 object should always remain in a consistent state. Write a driver program that tests the tick method, the incrementMinute method and the incrementHour method to ensure that they work correctly. Be sure to test the following cases:
   a) incrementing into the next minute.
   b) incrementing into the next hour.
   c) incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

8.5 Modify the Date class of Fig. 8.13 to perform error-checking on the initializer values for instance variables month, day and year (currently it validates only the month and day). Also, provide a method nextDay to increment the day by one. The Date object should always remain in a consistent state. Write a driver program that tests the nextDay method in a loop that prints the date during each iteration of the loop to illustrate that the nextDay method works correctly. Be sure to test the following cases:
   a) incrementing into the next month.
   b) incrementing into the next year.
8.6 Combine the modified `Time3` class of Exercise 8.5 and the modified `Date` class of Exercise 8.5 into one class called `DateAndTime`. Modify the `tick` method to call the `nextDay` method if the time is incremented into the next day. Modify methods `toString` and `toUniversalString()` to output the date in addition to the time. Write a driver program to test the new class `DateAndTime`. Specifically test incrementing the time to the next day.

8.7 Modify the set methods in class `Time3` of Fig. 8.8 to return appropriate error values if an attempt is made to set one of the instance variables `hour`, `minute` or `second` of an object of class `Time` to an invalid value. (Hint: Use `boolean` return types on each method.)

8.8 Create a class `Rectangle`. The class has attributes `length` and `width`, each of which defaults to 1. It has methods that calculate the `perimeter` and the `area` of the rectangle. It has `set` and `get` methods for both `length` and `width`. The `set` methods should verify that `length` and `width` are each floating-point numbers larger than 0.0 and less than 20.0. Write a program to test class `Rectangle`.

8.9 Create a more sophisticated `Rectangle` class than the one you created in Exercise 8.8. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a `set` method that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single x- or y-coordinate larger than 20.0. The `set` method also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide methods to calculate the `length`, `width`, `perimeter` and `area`. The length is the larger of the two dimensions. Include a predicate method `isSquare` which determines whether the rectangle is a square. Write a program to test class `Rectangle`.

8.10 Modify the `Rectangle` class of Exercise 8.9 to include a `draw` method that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Use the methods of the `Graphics` class to help output the `Rectangle`. If you feel ambitious, you might include methods to scale the size of the rectangle, rotate it and move it around within the designated portion of the first quadrant.

8.11 Create a class `HugeInteger` which uses a 40-element array of digits to store integers as large as 40 digits each. Provide methods `inputHugeInteger`, `outputHugeInteger`, `addHugeIntegers` and `subtractHugeIntegers`. For comparing `HugeInteger` objects, provide methods `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` and `isLessThanOrEqualTo`—each of these is a “predicate” method that simply returns `true` if the relationship holds between the two `HugeInteger`s and returns `false` if the relationship does not hold. Provide a predicate method `isZero`. If you feel ambitious, also provide the method `multiplyHugeIntegers`, the method `divideHugeIntegers` and the method `modulusHugeIntegers`.

8.12 Create a class `TicTacToe` that will enable you to write a complete program to play the game of Tic-Tac-Toe. The class contains as private data a 3-by-3 double array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square; place a 2 wherever the second player moves. Each move must be to an empty square. After each move determine whether the game has been won and whether the game is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players automatically. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional Tic-Tac-Toe on a 4-by-4-by-4 board [Note: This is a challenging project that could take many weeks of effort!].

8.13 Explain the notion of package access in Java. Explain the negative aspects of package access as described in the text.

8.14 What happens when a return type, even `void`, is specified for a constructor?
8.15 Create a Date class with the following capabilities:
   a) Output the date in multiple formats such as
      
      MM/DD/YYYY
      June 14, 1992
      DDD YYYY
   
   b) Use overloaded constructors to create Date objects initialized with dates of the formats
      in part a).

   [Hint: You can compare Strings using method equals. Suppose you have two String references
      s1 and s2, if those Strings are equal, s1.equals( s2 ) returns true; otherwise, it returns
      false.]

8.16 Create class SavingsAccount. Use a static class variable to store the annualInterestRate
   for all account holders. Each object of the class contains a private instance variable
   savingsBalance indicating the amount the saver currently has on deposit. Provide method
   calculateMonthlyInterest to calculate the monthly interest by multiplying the savingsBalance
   by annualInterestRate divided by 12; this interest should be added to savingsBalance. Provide a static
   method modifyInterestRate that sets the annualInterestRate to a new value. Write a driver program
   to test class SavingsAccount. Instantiate two savingsAccount objects, saver1 and saver2, with
   balances of $2000.00 and $3000.00, respectively. Set annualInterestRate to 4%, then calculate the
   monthly interest and print the new balances for each of the savers. Then set the
   annualInterestRate to 5% and calculate the next month’s interest and print the new balances
   for each of the savers.

8.17 Create class IntegerSet. Each object of the class can hold integers in the range 0 through
   100. A set is represented internally as an array of booleans. Array element a[i] is true if integer
   i is in the set. Array element a[j] is false if integer j is not in the set. The no-argument constructor
   initializes a set to the so-called “empty set” (i.e., a set whose array representation contains all false
   values).

   Provide the following methods: Method unionOfIntegerSets creates a third set which is
   the set-theoretic union of two existing sets (i.e., an element of the third set’s array is set to true
   if that element is true in either or both of the existing sets; otherwise, the element of the third set
   is set to false). Method intersectionOfIntegerSets creates a third set which is the set-the-
   oretic intersection of two existing sets i.e., an element of the third set’s array is set to false
   if that element is false in either or both of the existing sets; otherwise, the element of the third set
   is set to true). Method insertElement inserts a new integer k into a set (by setting a[k] to true).
   Method deleteElement deletes integer m (by setting a[m] to false). Method setPrint
   prints a set as a list of numbers separated by spaces. Print only those elements that are present in
   the set. Print --- for an empty set. Method isEqualTo determines if two sets are equal. Write a
   program to test your IntegerSet class. Instantiate several IntegerSet objects. Test that all your
   methods work properly.

8.18 It would be perfectly reasonable for the Time1 class of Figure 8.1 to represent the time
   internally as the number of seconds since midnight rather than the three integer values
   hour, minute
   and second. Clients could use the same public methods and get the same results. Modify the
   Time1 class of Figure 8.1 to implement the Time1 as the number of seconds since midnight and
   show that there is no change visible to the clients of the class.

8.19 (Drawing Program) Create a drawing applet that randomly draws lines, rectangles and ovals.
   For this purpose, create a set of “smart” shape classes where objects of these classes know how to
   draw themselves if provided with a Graphics object that tells them where to draw (i.e., the applet’s
   Graphics object allows a shape to draw on the applet’s background). The class names should be
   MyLine, MyRect and MyOval.
The data for class MyLine should include \( x1, y1, x2 \) and \( y2 \) coordinates. Method drawLine method of class Graphics will connect the two points supplied with a line. The data for classes MyRect and MyOval should include an upper-left \( x \)-coordinate value, an upper-left \( y \)-coordinate value, a \( width \) (must be nonnegative) and a \( height \) (must be nonnegative). All data in each class must be private.

In addition to the data, each class should define at least the following public methods:

a) A constructor with no arguments that sets the coordinates to 0.

b) A constructor with arguments that sets the coordinates to the supplied values.

c) Set methods for each individual piece of data that allow the programmer to independently set any piece of data in a shape (e.g., if you have an instance variable \( x1 \), you should have a method setX1).

d) Get methods for each individual piece of data that allow the programmer to independently retrieve any piece of data in a shape (e.g., if you have an instance variable \( x1 \), you should have a method getX1).

e) A draw method with the first line

\[
\text{public void draw( Graphics g )}
\]

will be called from the applet’s paint method to draw a shape onto the screen.

The preceding methods are required. If you would like to provide more methods for flexibility, please do so.

Begin by defining class MyLine and an applet to test your classes. The applet should have a MyLine instance variable line that can refer to one MyLine object (created in the applet’s init method with random coordinates). The applet’s paint method should draw the shape with a statement like

\[
\text{line.draw( g );}
\]

where line is the MyLine reference and g is the Graphics object that the shape will use to draw itself on the applet.

Next, change the single MyLine reference into an array of MyLine references and hard code several MyLine objects into the program for drawing. The applet’s paint method should walk through the array of MyLine objects and draw every one.

After the preceding part is working, you should define the MyOval and MyRect classes and add objects of these classes into the MyRect and MyOval arrays. The applet’s paint method should walk through each array and draw every shape. Create five shapes of each type.

Once the applet is running, select Reload from the appletviewer’s Applet menu to reload the applet. This will cause the applet to choose new random numbers for the shapes and draw the shapes again.

In Chapter 9, we will modify this exercise to take advantage of the similarities between the classes and to avoid reinventing the wheel.
Object-Oriented Programming

Objectives

• To understand inheritance and software reusability.
• To understand superclasses and subclasses.
• To appreciate how polymorphism makes systems extensible and maintainable.
• To understand the distinction between abstract classes and concrete classes.
• To learn how to create abstract classes and interfaces.

Say not you know another entirely, till you have divided an inheritance with him.
Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
Bertrand Russell

Good as it is to inherit a library, it is better to collect one.
Augustine Birrell

General propositions do not decide concrete cases.
Oliver Wendell Holmes

A philosopher of imposing stature doesn’t think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
Alfred North Whitehead
9.1 Introduction

In this chapter, we discuss object-oriented programming (OOP) and its key component technologies—inheritance and polymorphism. Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and adding new capabilities the new classes require. Inheritance takes advantage of class relationships where objects of a certain class—such as a class of vehicles—have the same characteristics. Newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class “convertible” certainly has the characteristics of the more general class “automobile,” but a convertible’s roof goes up and down.
Software reusability saves time in program development. It encourages reuse of proven and debugged high-quality software, thus reducing problems after a system becomes operational. These are exciting possibilities. Polymorphism enables us to write programs in a general fashion to handle a wide variety of existing and yet-to-be-specified related classes. Polymorphism makes it easy to add new capabilities to a system. Inheritance and polymorphism are effective techniques for dealing with software complexity.

When creating a new class, instead of writing completely new instance variables and instance methods, the programmer can designate that the new class is to inherit the instance variables and instance methods of a previously defined superclass. The new class is referred to as a subclass. Each subclass itself becomes a candidate to be a superclass for some future subclass.

The direct superclass of a class is the superclass from which the class explicitly inherits (via the keyword extends). An indirect superclass is inherited from two or more levels up the class hierarchy. For example, class JApplet (package javax.swing) extends class Applet (package java.applet). Thus, each applet class we have defined is a direct subclass of JApplet and an indirect subclass of Applet.

With single inheritance, a class is derived from one superclass. Java does not support multiple inheritance (as C++ does) but it does support the notion of interfaces. Interfaces help Java achieve many of the advantages of multiple inheritance without the associated problems. We will discuss the details of interfaces in this chapter. We consider both general principles and a detailed specific example of creating and using interfaces.

A subclass normally adds instance variables and instance methods of its own, so a subclass is generally larger than its superclass. A subclass is more specific than its superclass and represents a smaller, more specialized group of objects. With single inheritance, the subclass starts out essentially the same as the superclass. The real strength of inheritance comes from the ability to define in the subclass additions to, or replacements for, the features inherited from the superclass.

Every subclass object is also an object of that class’s superclass. For example, every applet we have defined is considered to be an object of class JApplet. Also, because JApplet extends Applet, every applet we have defined is considered to be an Applet. This information is critical when developing applets, because an applet container can execute a program only if it is an Applet. Although a subclass object always can be treated as one of its superclass types, superclass objects are not considered to be objects of their subclass types. We will take advantage of this “subclass-object-is-a-superclass-object” relationship to perform some powerful manipulations. For example, a drawing application can maintain a list of shapes to display. If all the shape types extend the same superclass directly or indirectly, the drawing program can store all the shapes in an array (or other data structure) of superclass objects. As we will see in this chapter, this ability to process a set of objects as a single type is a key thrust of object-oriented programming.

We add a new form of member access control in this chapter, namely protected access. Subclass methods and methods of other classes in the same package as the superclass can access protected superclass members.

Experience in building software systems indicates that significant portions of the code deal with closely related special cases. It becomes difficult in such systems to see the “big picture” because the designer and the programmer become preoccupied with the special cases. Object-oriented programming provides several ways of “seeing the forest through the trees.”
The programmer and designer concentrate on the big picture—the commonality among objects in the system—rather than the special cases. This process is called abstraction.

If a procedural program has many closely related special cases, then it is common to see switch structures or nested if/else structures that distinguish among the special cases and provide the processing logic to deal with each case individually. We will show how to use inheritance and polymorphism to replace such switch logic with much simpler logic.

We distinguish between the “is a” relationship and the “has a” relationship. “Is a” is inheritance. In an “is a” relationship, an object of a subclass type may also be treated as an object of its superclass type. “Has a” is composition (as we discussed in Chapter 8). In a “has a” relationship, a class object has one or more objects of other classes as members. For example, a car has a steering wheel.

A subclass’ methods might need to access certain of its superclass’ instance variables and methods. A crucial aspect of software engineering in Java is that a subclass cannot access the private members of its superclass. If a subclass could access the superclass’s private members, this would violate information hiding in the superclass.

Software Engineering Observation 9.1

A subclass cannot directly access private members of its superclass.

Testing and Debugging Tip 9.1

Hiding private members is a tremendous help in testing, debugging and correctly modifying systems. If a subclass could access its superclass’s private members, it would be possible for classes derived from that subclass to access that data as well, and so on. This would propagate access to what is supposed to be private data, and the benefits of information hiding would be lost throughout the class hierarchy.

However, a subclass can access the public and protected members of its superclass. A subclass also can use the package access members of its superclass if the subclass and superclass are in the same package. Superclass members that should not be accessible to a subclass via inheritance are declared private in the superclass. A subclass can effect state changes in superclass private members only through public, protected and package access methods provided in the superclass and inherited into the subclass. [Note: We use protected instance variables in this chapter to demonstrate how they work. Several of the exercises in this chapter require that you use only private instance variables, to maintain encapsulation.]

Software Engineering Observation 9.2

To preserve encapsulation, all instance variables should be declared private and should be accessible only via set and get methods of the class.

A problem with inheritance is that a subclass can inherit methods that it does not need or should not have. It is the class designer’s responsibility to ensure that the capabilities provided by a class are appropriate for future subclasses. Even when the superclass methods are appropriate for the subclasses, it is common for a subclass to require the method to perform a task in a manner that is specific to the subclass. In such cases, the superclass method can be overridden (redefined) in the subclass with an appropriate implementation.

Perhaps most exciting is the notion that new classes can inherit from abundant class libraries, such as those provided with the Java API. Organizations develop their own class
libraries and can take advantage of other libraries available worldwide. Someday, most software might be constructed from *standardized reusable components*, just as hardware is often constructed today. This will help meet the challenges of developing the ever more powerful software we will need in the future.

### 9.2 Superclasses and Subclasses

Often an object of one class “is an” object of another class as well. A rectangle certainly *is a* quadrilateral (as are squares, parallelograms and trapezoids). Thus, class *Rectangle* can be said to *inherit* from class *Quadrilateral*. In this context, class *Quadrilateral* is a superclass, and class *Rectangle* is a subclass. A rectangle *is a* specific type of quadrilateral, but it is incorrect to claim that a quadrilateral *is a* rectangle (the quadrilateral could be a parallelogram). Figure 9.1 shows several simple inheritance examples of superclasses and potential subclasses.

Inheritance normally produces subclasses with *more* features than their superclasses, so the terms superclass and subclass can be confusing. There is another way, however, to view these terms that makes perfectly good sense. Because every subclass object “is an” object of its superclass, and because one superclass can have many subclasses, the set of objects represented by a superclass is normally larger than the set of objects represented by any of that superclass’s subclasses. For example, the superclass *Vehicle* represents in a generic manner all vehicles, such as cars, trucks, boats, bicycles and so on. However, subclass *Car* represents only a small subset of all the *Vehicles* in the world.

Inheritance relationships form tree-like hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. A class can certainly exist by itself, but it is when a class is used with the mechanism of inheritance that the class becomes either a superclass that supplies attributes and behaviors to other classes or a subclass that inherits those attributes and behaviors. Frequently, one class is both a subclass and a superclass.

<table>
<thead>
<tr>
<th>Superclass</th>
<th>Subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent</td>
</tr>
<tr>
<td></td>
<td>UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle</td>
</tr>
<tr>
<td></td>
<td>Triangle</td>
</tr>
<tr>
<td></td>
<td>Rectangle</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan</td>
</tr>
<tr>
<td></td>
<td>HomeImprovementLoan</td>
</tr>
<tr>
<td></td>
<td>MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>FacultyMember</td>
</tr>
<tr>
<td></td>
<td>StaffMember</td>
</tr>
<tr>
<td>Account</td>
<td>CheckingAccount</td>
</tr>
<tr>
<td></td>
<td>SavingsAccount</td>
</tr>
</tbody>
</table>

**Fig. 9.1** Some simple inheritance examples in which the subclass “is a” superclass.
Let us develop a simple inheritance hierarchy. A typical university community has thousands of people who are community members. These people consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teaching faculty. This yields the inheritance hierarchy shown in Fig. 9.2. Note that the inheritance hierarchy could contain many other classes. For example, students can be graduate students or undergraduate students. Undergraduate students can be freshman, sophomores, juniors, and seniors. And so on. The arrows in the hierarchy represent the “is a” relationship. For example, based on this class hierarchy that we can state, “an Employee is a CommunityMember,” or “a Teacher is a Faculty member.”

CommunityMember is the direct superclass of Employee, Student and Alumni. CommunityMember is an indirect superclass of all the other classes in the hierarchy diagram. Note that class Employee is both a subclass of CommunityMember and a superclass of Faculty and Staff.

Also, starting from the bottom of the diagram, you can follow the arrows and apply the is a relationship all the way up to the topmost superclass in the hierarchy. For example, an Administrator is a Faculty member, is an Employee and is a CommunityMember. And, in Java, an Administrator also is an Object because all classes in Java have Object as one of their direct or indirect superclasses. Thus, all classes in Java are related in a hierarchical relationship in which they share the 11 methods defined by class Object, which include the toString and finalize methods discussed previously. Other methods of class Object are discussed as they are needed in the text.

Another substantial inheritance hierarchy is the Shape hierarchy of Figure 9.3. There are abundant examples of hierarchies in the real world, but students are not accustomed to categorizing the real world in this manner, so it takes some adjustment in their thinking. Actually, biology students have had some practice with hierarchies. Everything we study in biology is grouped into a hierarchy headed by living things and these can be plants or animals and so on.

To specify that class TwoDimensionalShape is derived from (or inherits from) class Shape, class TwoDimensionalShape could be defined in Java as follows:
public class TwoDimensionalShape extends Shape {
    ...
}

With inheritance, private members of a superclass are not directly accessible from that class’s subclasses. Package access members of the superclass are accessible in a subclass only if both the superclass and its subclass are in the same package. All other superclass members become members of the subclass, using their original member access (i.e., public members of the superclass become public members of the subclass, and protected members of the superclass become protected members of the subclass).

Software Engineering Observation 9.3

Constructors are never inherited—they are specific to the class in which they are defined.

It is possible to treat superclass objects and subclass objects similarly; that commonality is expressed in the attributes and behaviors of the superclass. Objects of all classes derived from a common superclass can all be treated as objects of that superclass.

We will consider many examples in which we can take advantage of this inheritance relationship with an ease of programming not available in non-object-oriented languages such as C.

9.3 protected Members

A superclass’s public members are accessible anywhere the program has a reference to that superclass type or one of its subclass types. A superclass’s private members are accessible only in methods of that superclass.

A superclass’s protected access members serve as an intermediate level of protection between public and private access. A superclass’s protected members may be accessed only by methods of the superclass, by methods of subclasses and by methods of other classes in the same package (protected members have package access).

Subclass methods can normally refer to public and protected members of the superclass simply by using the member names. When a subclass method overrides a superclass method, the superclass method may be accessed from the subclass by preceding the superclass method name with keyword super followed by the dot operator (.). This technique is illustrated several times throughout the chapter.
9.4 Relationship between Superclass Objects and Subclass Objects

An object of a subclass can be treated as an object of its superclass. This makes possible some interesting manipulations. For example, despite the fact that objects of a variety of classes derived from a particular superclass might be quite different from one another, we can create an array of references to them—as long as we treat them as superclass objects. But the reverse is not true: A superclass object cannot always be treated a subclass object. For example, a Shape is not always a Circle.

However, an explicit cast can be used to convert a superclass reference to a subclass reference. This can be done only when the superclass reference is referencing a subclass object; otherwise, Java will indicate a ClassCastException—an indication that the cast operation is not allowed. Exceptions are discussed in detail in Chapter 14.

Common Programming Error 9.1
Assigning an object of a superclass to a subclass reference (without a cast) is a syntax error.

Software Engineering Observation 9.4
If an object has been assigned to a reference of one of its superclasses, it is acceptable to cast that object back to its own type. In fact, this must be done in order to send that object any of its messages that do not appear in that superclass.

Our first example consists of two classes. Figure 9.4 shows a Point class definition. Figure 9.5 shows a Circle class definition. We will see that class Circle inherits from class Point. Figure 9.6 shows application class InheritanceTest, which demonstrates assigning subclass references to superclass references and casting superclass references to subclass references.

Every applet defined previously has used some of the techniques presented here. We now formalize the inheritance concept. In Chapter 3, we stated that every class definition in Java must extend another class. However, notice in Fig. 9.4 that class Point (line 4) does not use the extends keyword explicitly. If a new class definition does not extend an existing class definition explicitly, Java implicitly uses class Object (package java.lang) as the superclass for the new class definition. Class Object provides a set of methods that can be used with any object of any class.

Software Engineering Observation 9.5
Every class in Java implicitly extends Object, unless it is specified otherwise in the first line of the class definition, in which case the class indirectly extends Object. Thus, class Object is the superclass of the entire Java class hierarchy.

Let us first examine the Point class definition (Fig. 9.4). The public services of class Point include methods setPoint, getX, getY, toString and two Point constructors. The instance variables x and y of Point are specified as protected. This prevents clients of Point objects from accessing the data directly (unless they are classes in the same package), but enables classes derived from Point to access the inherited instance variables directly. If the data were specified as private, the non-private methods of Point would have to be used to access the data, even by subclasses. Note that class Point’s toString method overrides the original toString method inherited from class Object.
Class **Point**’s constructors (lines 8–12 and 15–19) must call class **Object**’s constructor. In fact, every subclass constructor is required to call its direct superclass’s constructor as its first task, either implicitly or explicitly (the syntax for this call is discussed with class **Circle** momentarily). If there is no explicit call to the superclass constructor,
Java automatically attempts to call the superclass’s default constructor. Note that lines 10 and 17 are comments indicating where the call to the superclass `Object`’s default constructor occurs.

**Software Engineering Observation 9.6**

Every subclass constructor must call one of the direct superclass constructors explicitly or implicitly. Implicit calls can be made only to the no-argument constructor of the superclass. If the superclass does not provide a no-argument constructor, all direct subclasses of that class must call one of superclass’s constructors explicitly.

Class `Circle` (Fig. 9.5) inherits from class `Point` as specified with the `extends` keyword on line 4. Keyword `extends` in the class definition indicates inheritance. All the (non-private) members of class `Point` (except the constructors) are inherited into class `Circle`. Thus, the public interface to `Circle` includes the `Point` class’s public methods as well as the two overloaded `Circle` constructors and `Circle` methods `setRadius`, `getRadius`, `area` and `toString`. Notice that method `area` (lines 38–41) uses predefined constant `Math.PI` from class `Math` (package `java.lang`) to calculate the area of a circle.

```java
// Fig. 9.5: Circle.java
// Definition of class Circle

public class Circle extends Point {  // inherits from Point
    protected double radius;

    // no-argument constructor
    public Circle()
    {
        // implicit call to superclass constructor occurs here
        setRadius( 0 );
    }

    // constructor
    public Circle( double circleRadius, int xCoordinate, int yCoordinate )
    {
        // call superclass constructor to set coordinates
        super( xCoordinate, yCoordinate );

        // set radius
        setRadius( circleRadius );
    }

    // set radius of Circle
    public void setRadius( double circleRadius )
    {
        radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
    }

Fig. 9.5  Circle class definition (part 1 of 2).
```
The **Circle** constructors (lines 8–12 and 15–23) must invoke a **Point** constructor to initialize the superclass portion of a **Circle** object (i.e., variables **x** and **y** inherited from **Point**). The default constructor at lines 8–12 does not call a **Point** constructor explicitly, so Java automatically calls class **Point**'s default constructor (defined at line 8 of Fig. 9.4) that initializes superclass members **x** and **y** to zeros. If class **Point** contained only the constructor with arguments (i.e., did not provide a default constructor), a compiler error would occur.

Line 19 in the body of the second **Circle** constructor explicitly invokes the **Point** constructor (defined at line 15 of Fig. 9.4) by using the superclass constructor call syntax—keyword `super` followed by a set of parentheses containing the arguments to the superclass constructor. In this case, the arguments are the values **xCordinate** and **yCoor-dinate** that are used by the **Point** constructor to initialize the superclass members **x** and **y**. The call to the superclass constructor must be the first line in the body of the subclass constructor. To call the superclass default constructor explicitly, use the statement

```java
super();  // explicit call to superclass default constructor
```

### Common Programming Error 9.2

It is a syntax error if a `super` call by a subclass to its superclass constructor is not the first statement in the subclass constructor.

### Common Programming Error 9.3

It is a syntax error if the arguments to a `super` call by a subclass to its superclass constructor do not match the parameters specified in one of the superclass constructor definitions.

A subclass can redefine a superclass method by using the same signature; this is called *overriding* a superclass method. When that method is mentioned by name in the subclass,
the subclass version is automatically called. We have actually been overriding methods in every applet in the book. When we extend JApplet to create a new applet class, the new class inherits versions of init and paint (and many other methods). Each time we defined init or paint, we were overriding the original version that was inherited. Also, when we provided method toString for many of the classes in Chapter 8, we were overriding the original version of toString provided by class Object. As we will see in Fig. 9.8, the super reference followed by the dot operator can be used to access the original superclass version of that method from the subclass.

Note that class Circle’s toString method (lines 44–48) overrides the Point class toString method (lines 41–44 of Fig. 9.4). Class Point’s toString method overrides the original toString method provided by class Object. Actually, every class inherits a toString method, because class Object provides the original toString method. This method converts an object of any class into a String representation and is sometimes called implicitly by the program (e.g., when an object is concatenated with a String). Circle method toString directly accesses the protected instance variables x and y that were inherited from class Point. Method toString uses the values of x and y as part of the Circle’s String representation. Actually, if you study class Point’s toString method and class Circle’s toString method, you will notice that Circle’s toString uses the same formatting as Point’s toString for the Point parts of the Circle. Also, recall Software Engineering Observation 8.14, indicating that, if a method exists that performs part of another method’s task, call the method. Point’s toString performs part of the task of Circle’s toString. To call Point’s toString from class Circle, use the expression

```
super.toString()
```

Software Engineering Observation 9.7

A redefinition of a superclass method in a subclass need not have the same signature as the superclass method. Such a redefinition is not method overriding; rather, it is an example of method overloading.

Software Engineering Observation 9.8

Any object can be converted to a String with an explicit or implicit call to the object’s toString method.

Software Engineering Observation 9.9

Each class should override method toString to return useful information about objects of that class.

Common Programming Error 9.4

It is a syntax error if a method in a superclass and a method in its subclass have the same signature but a different return type.

The InheritanceTest application (Fig. 9.6) instantiates Point object point1 and Circle object circle1 at lines 18–19 in main. The String representations of each object are assigned to String output to show that they were initialized correctly (lines 21–22). See the first two lines in the output screen capture to confirm this.
// Fig. 9.6: InheritanceTest.java
// Demonstrating the "is a" relationship

// Java core packages
import java.text.DecimalFormat;

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

public class InheritanceTest {

    public static void main( String args[] )
    {
        Point point1, point2;
        Circle circle1, circle2;

        point1 = new Point( 30, 50 );
        circle1 = new Circle( 2.7, 120, 89 );

        String output = "Point point1: " + point1.toString() + 
                        "\nCircle circle1: " + circle1.toString();

        // use "is a" relationship to refer to a Circle
        // with a Point reference
        point2 = circle1;   // assigns Circle to a Point reference

        output += "\n
Circle circle1 (via point2 reference): " + 
                  point2.toString();

        // use downcasting (casting a superclass reference to a
        // subclass data type) to assign point2 to circle2
        circle2 = ( Circle ) point2;

        output += "\n\nCircle circle1 (via circle2): " + 
                  circle2.toString();

        DecimalFormat precision2 = new DecimalFormat( "0.00" );
        output += "\nArea of c (via circle2): " + 
                  precision2.format( circle2.area() );

        // attempt to refer to Point object with Circle reference
        if ( point1 instanceof Circle ) {
            circle2 = ( Circle ) point1;
            output += "\ncast successful";
        }
        else
            output += "\npoint1 does not refer to a Circle";

        JOptionPane.showMessageDialog( null, output,
                                       "Demonstrating the "is a" relationship",
                                       JOptionPane.INFORMATION_MESSAGE );
    }
}

Fig. 9.6 Assigning subclass references to superclass references (part 1 of 2).
© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
Line 26 assigns `circle1` (a reference to a subclass `Circle` object) to `point2` (a superclass `Point` reference). It is always acceptable in Java to assign a subclass reference to a superclass reference, because of the “is a” relationship of inheritance. A `Circle` is a `Point` because class `Circle` extends class `Point`. Assigning a superclass reference to a subclass reference is dangerous, as we will see.

Lines 28–29 append the result of `point2.toString()` to `output`. Interestingly, when `point2` is sent the `toString` message, Java knows that the object really is a `Circle`, so it chooses the `Circle` class’s `toString` rather than the `Point` class’s `toString` as you might have expected. This is an example of polymorphism and dynamic binding—concepts we treat in depth later in this chapter. The compiler looks at the preceding expression and asks the question, “Does the data type of the reference `point2` (i.e., `Point`) have a `toString` method with no arguments?” The answer to this question is yes (per `Point`’s `toString` definition on line 41 of Fig. 9.4). The compiler simply checks the syntax of the expression and ensures that the method exists. At execution time, the interpreter asks the question, “What type is the object to which `point2` refers?” Every object in Java knows its own data type, so the answer to the question is `point2` refers to a `Circle` object. Based on this answer, the interpreter calls the `toString` method of the actual object’s data type—class `Circle`’s `toString` method. See the third line of the screen capture to confirm this. The two key programming techniques we used to achieve this polymorphism effect are

1. extending class `Point` to create class `Circle`, and
2. overriding method `toString` with the exact same signature in class `Point` and class `Circle`.

Line 33 casts `point2`, which admittedly is referencing a `Circle` at this time in the program’s execution, to a `Circle` and assigns the result to `circle2` (this cast would be dangerous if `point2` were really referencing a `Point`, as we will soon discuss). Then we use `circle2` to append to `output` the various facts about `circle2`. Lines 35–36
invoke method **toString** to append the **String** representation of the **Circle**. Lines 39–40 append the **area** of the **Circle** to **output**.

Next, the **if/else** structure at lines 43–48 attempts a dangerous cast in line 44. We cast **point1**, which refers to a **Point** object, to a **Circle**. If the program attempts to execute this statement, Java would determine that **point1** really refers to a **Point**, recognize the cast to **Circle** as being dangerous and indicate an improper cast with **ClassCastException** message. However, we prevent this statement from executing with the **if** condition

```java
if ( point1 instanceof Circle ) {
```

that uses operator **instanceof** to determine whether the object to which **point1** refers **is a Circle**. This condition evaluates to **true** only if the object to which **point1** refers **is a Circle**; otherwise, the condition evaluates to **false**. Reference **point1** does not refer to a **Circle**, so the condition fails, and a **String** indicating that **point1** does not refer to a **Circle** is appended to **output**.

If we remove the **if** test from the program and execute the program, the following message is generated at execution time:

```
Exception in thread "main" java.lang.ClassCastException: Point
at InheritanceTest.main(InheritanceTest.java:43)
```

Such error messages normally include the file name (**InheritanceTest.java**) and line number at which the error occurred (43) so you can go to that specific line in the program for debugging.

### 9.5 Constructors and Finalizers in Subclasses

When an object of a subclass is instantiated, the superclass’s constructor should be called to do any necessary initialization of the superclass instance variables of the subclass object. An explicit call to the superclass constructor (via the **super** reference) can be provided as the first statement in the subclass constructor. Otherwise, the subclass constructor will call the superclass default constructor (or no-argument constructor) implicitly.

Superclass constructors are not inherited by subclasses. Subclass constructors, however, can call superclass constructors via the **super** reference.

**Software Engineering Observation 9.10**

*When an object of a subclass is created, first the subclass constructor calls the superclass constructor (explicitly via super or implicitly), the superclass constructor executes, then the remainder of the subclass constructor’s body executes.*

If the classes in your class hierarchy define **finalize** methods, the subclass **finalize** method as its last action should invoke the superclass **finalize** method to ensure that all parts of an object are finalized properly if the garbage collector reclaims the memory for the object.

The application of Fig. 9.7–Fig. 9.9 shows the order in which superclass and subclass constructors and finalizers are called. For the purpose of this example, class **Point** and class **Circle** are simplified.

Class **Point** (Fig. 9.7) contains two constructors, a finalizer, a **toString** method and **protected** instance variables **x** and **y**. The constructor and finalizer each print that they are executing, then display the **Point** for which they are invoked. Note the use of
this in the `System.out.println` calls to cause an implicit call to method `toString`. Notice the first line of the `finalize` method (line 23). Method `finalize` should always be defined as `protected` so subclasses have access to the method but classes that simply use `Point` objects do not.

Class `Circle` (Fig. 9.8) derives from `Point` and contains two constructors, a finalizer, a `toString` method and protected instance variable `radius`. The constructor and finalizer each print that they are executing, then display the `Circle` for which they are invoked. Note that the `Circle` method `toString` invokes `Point`’s `toString` via `super` (line 19).

```
// Fig. 9.7: Point.java
// Definition of class Point
public class Point extends Object {

  // coordinates of the Point
  protected int x, y;

  // no-argument constructor
  public Point() {
    x = 0;
    y = 0;
    System.out.println( "Point constructor: " + this );
  }

  // constructor
  public Point( int xCoordinate, int yCoordinate ) {
    x = xCoordinate;
    y = yCoordinate;
    System.out.println( "Point constructor: " + this );
  }

  // finalizer
  protected void finalize() {
    System.out.println( "Point finalizer: " + this );
  }

  // convert Point into a String representation
  public String toString() {
    return "[" + x + ", " + y + "]";
  }
}
```

Fig. 9.7 `Point` class definition to demonstrate when constructors and finalizers are called.

Software Engineering Observation 9.11

When a superclass method is overridden in a subclass, it is common to have the subclass version call the superclass version and do some additional work. In this scenario, the superclass method performs the tasks common to all subclasses of that class, and the subclass method performs additional tasks specific to a given subclass.
Common Programming Error 9.5

When an overridden method calls the superclass version of the same method, not using keyword super to reference the superclass’s method causes infinite recursion, because the subclass method actually calls itself.

Common Programming Error 9.6

Cascading super reference to refer to a member (method or variable) several levels up the hierarchy (as in super.super.x) is a syntax error.
Application class Test (Fig. 9.9) uses this Point/Circle inheritance hierarchy. The application begins in method main by instantiating Circle object circle1 (line 11). This invokes the Circle constructor at line 15 of Fig. 9.8, which immediately invokes the Point constructor at line 15 of Fig. 9.7. The Point constructor outputs the values received from the Circle constructor by implicitly calling method toString and returns program control to the Circle constructor. Then the Circle constructor outputs the complete Circle by calling method toString. Notice that the first two lines of the output from this program both show values for \( x, y \) and radius. Polymorphism is once again causing the Circle's toString method to execute because it is a Circle object that is being created. When toString is invoked from the Point constructor, 0.0 is displayed for the radius because the radius has not yet been initialized in the Circle constructor.

Circle object circle2 is instantiated next. Again, the Point and Circle constructors both execute. Notice, in the command-line output window, that the body of the Point constructor is performed before the body of the Circle constructor, showing that objects are constructed “inside out.”

```java
public class Test {

    public static void main( String args[] )
    {
        Circle circle1, circle2;

        circle1 = new Circle( 4.5, 72, 29 );
        circle2 = new Circle( 10, 5, 5 );

        circle1 = null; // mark for garbage collection
        circle2 = null; // mark for garbage collection

        System.gc();     // call the garbage collector
    }

} // end class Test
```

**Fig. 9.9**  Order in which constructors and finalizers are called.
Lines 14–15 set `circle1` to null, then set `circle2` to null. Each of these objects is no longer needed in the program, so Java marks the memory occupied by `circle1` and `circle2` for garbage collection. Java guarantees that, before the garbage collector runs to reclaim the space for each of these objects, the `finalize` methods for each object will be called. The garbage collector is a low-priority thread that runs automatically whenever processor time is available. We choose here to ask the garbage collector to run with a call to class `System`’s static method `gc` in line 17. Java does not guarantee the order in which objects will be garbage collected; therefore, it cannot guarantee which object’s finalizer will execute first. Notice, in the command-line output window, that `finalize` methods are called for both the `Circle` and `Point` when each `Circle` object is garbage collected.

### 9.6 Implicit Subclass-Object-to-Superclass-Object Conversion

Despite the fact that a subclass object also “is a” superclass object, the subclass type and the superclass type are different. Subclass objects can be treated as superclass objects. This makes sense because the subclass has members corresponding to each of the superclass members—remember that the subclass normally has more members than the superclass has. Assignment in the other direction is not allowed because assigning a superclass object to a subclass reference would leave the additional subclass members undefined.

A reference to a subclass object could be implicitly converted into a reference to a superclass object because a subclass object is a superclass object through inheritance.

There are four possible ways to mix and match superclass references and subclass references with superclass objects and subclass objects:

1. Referring to a superclass object with a superclass reference is straightforward.
2. Referring to a subclass object with a subclass reference is straightforward.
3. Referring to a subclass object with a superclass reference is safe, because the subclass object is an object of its superclass as well. Such code can refer only to superclass members. If this code refers to subclass-only members through the superclass reference, the compiler will report a syntax error.
4. Referring to a superclass object with a subclass reference is a syntax error.

As convenient as it might be to treat subclass objects as superclass objects, and to do this by manipulating all these objects with superclass references, there appears to be a problem. In a payroll system, for example, we would like to be able to walk through an array of employees and calculate the weekly pay for each person. But intuition suggests that using superclass references would enable the program to call only the superclass payroll calculation routine (if indeed there is such a routine in the superclass). We need a way to invoke the proper payroll calculation routine for each object, whether it is a superclass object or a subclass object, and to do this simply by using the superclass reference. Actually, this is precisely how Java behaves and is discussed in this chapter when we consider polymorphism and dynamic binding.
9.7 Software Engineering with Inheritance

We can use inheritance to customize existing software. When we use inheritance to create a new class from an existing class, the new class inherits the attributes and behaviors of an existing class; then we can add attributes and behaviors or override superclass behaviors to customize the class to meet our needs.

It can be difficult for students to appreciate the problems faced by designers and implementers on large-scale software projects in industry. People experienced on such projects will invariably state that a key to improving the software development process is encouraging software reuse. Object-oriented programming in general, and Java in particular, certainly does this.

It is the availability of substantial and useful class libraries that delivers the maximum benefits of software reuse through inheritance. As interest in Java grows, interest in Java class libraries will increase. Just as shrink-wrapped software produced by independent software vendors became an explosive growth industry with the arrival of the personal computer, so, too, will the creation and sale of Java class libraries. Application designers will build their applications with these libraries, and library designers will be rewarded by having their libraries wrapped with the applications. What we see coming is a massive worldwide commitment to the development of Java class libraries for a huge variety of applications arenas.

Software Engineering Observation 9.12
Creating a subclass does not affect its superclass's source code or the superclass's Java bytecodes; the integrity of a superclass is preserved by inheritance.

A superclass specifies commonality. All classes derived from a superclass inherit the capabilities of that superclass. In the object-oriented design process, the designer looks for commonality among a set of classes and factors it out to form desirable superclasses. Subclasses are then customized beyond the capabilities inherited from the superclass.

Software Engineering Observation 9.13
Just as the designer of non-object-oriented systems should avoid unnecessary proliferation of functions, the designer of object-oriented systems should avoid unnecessary proliferation of classes. Proliferating classes creates management problems and can hinder software reusability, simply because it is more difficult for a potential user of a class to locate that class in a huge collection. The trade-off is to create fewer classes, each providing substantial additional functionality, but such classes might be too rich for certain users.

Performance Tip 9.1
When creating a new class, inherit from the class “closest” to what you need—i.e., the one that provides the minimum set of capabilities required for a new class to perform its tasks. Subclasses could inherit data and functionality that they will not use, in which case memory and processing resources may be wasted.

Note that reading a set of subclass declarations can be confusing because inherited members are not shown, but inherited members are nevertheless present in the subclasses. A similar problem can exist in the documentation of subclasses.

Software Engineering Observation 9.14
In an object-oriented system, classes are often closely related. “Factor out” common attributes and behaviors and place these in a superclass. Then use inheritance to form subclasses without having to repeat common attributes and behaviors.
Software Engineering Observation 9.15

Modifications to a superclass do not require subclasses to change as long as the public interface to the superclass remains unchanged.

9.8 Composition vs. Inheritance

We have discussed is a relationships that are implemented by inheritance. We have also discussed has a relationships (and seen examples in preceding chapters) in which a class has objects of other classes as members—such relationships create new classes by composition of existing classes. For example, given the classes Employee, BirthDate and TelephoneNumber, it is improper to say that an Employee is a BirthDate or that an Employee is a TelephoneNumber. But it is certainly appropriate to say that an Employee has a BirthDate and that an Employee has a TelephoneNumber.

9.9 Case Study: Point, Circle, Cylinder

Now let us consider a substantial inheritance example. We consider a point, circle, cylinder hierarchy. First we develop and use class Point (Fig. 9.10 and Fig. 9.11). Then we present an example in which we derive class Circle from class Point (Fig. 9.12 and Fig. 9.13). Finally, we present an example in which we derive class Cylinder from class Circle (Fig. 9.14 and Fig. 9.15).

Figure 9.10 is the class Point definition. Class Point is defined as part of package com.deitel.jhtp4.ch09 (line 3). Note that Point’s instance variables are protected. Thus, when class Circle is derived from class Point, the methods of class Circle will be able to reference coordinates x and y directly rather than using access methods. This could result in better performance.

```
// Fig. 9.10: Point.java
// Definition of class Point
package com.deitel.jhtp4.ch09;

public class Point {
    protected int x, y; // coordinates of Point

    // No-argument constructor
    public Point() {
        // implicit call to superclass constructor occurs here
        setPoint( 0, 0 );
    }

    // constructor
    public Point( int xCoordinate, int yCoordinate ) {
        // implicit call to superclass constructor occurs here
        setPoint( xCoordinate, yCoordinate );
    }
}
```

Fig. 9.10 Point class definition (part 1 of 2).
Figure 9.11 shows a Test application for testing class Point. The main method must use \texttt{getX} and \texttt{getY} to read the values of \texttt{protected} instance variables \texttt{x} and \texttt{y}. Remember that \texttt{protected} instance variables are accessible only to methods of their class, their subclasses and other classes in the same package. Also, note the implicit call to \texttt{toString} when \texttt{point} is added to a \texttt{String} at line 25.
Our next example imports the `Point` class definition from Fig. 9.10, so we do not show the class definition again here. Figure 9.12 shows the `Circle` class definition with the `Circle` method definitions. Note that class `Circle` extends class `Point`. This means that the public interface to `Circle` includes the `Point` methods as well as the `Circle` methods `setRadius`, `getRadius`, `area`, `toString` and the `Circle` constructors.

```java
// Fig. 9.12: Circle.java
// Definition of class Circle
package com.deitel.jhtp4.ch09;

public class Circle extends Point {  // inherits from Point
    protected double radius;

    // no-argument constructor
    public Circle() { // implicit call to superclass constructor occurs here
        setRadius( 0 );
    }

    // get coordinates
    String output = "X coordinate is " + point.getX() +
                    "\nY coordinate is " + point.getY();

    // set coordinates
    point.setPoint( 10, 10 );

    // use implicit call to point.toString()
    output += "\n\nThe new location of point is " + point;

    JOptionPane.showMessageDialog( null, output,
                    "Demonstrating Class Point",
                    JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
}
```
Object-Oriented Programming

Application Test (Fig. 9.13) instantiates an object of class Circle (line 19), then uses get methods to obtain the information about the Circle object. Method main indirectly references the protected data of class Circle through method calls. Method main then uses set methods setRadius and setPoint to reset the radius and coordinates of the center of the circle. Finally, main displays the Circle object circle and calculates and displays its area.
import java.text.DecimalFormat;

import javax.swing.JOptionPane;

import com.deitel.jhtp4.ch09.Circle;

public class Test {
  public static void main( String args[] ) {
    // create a Circle
    Circle circle = new Circle( 2.5, 37, 43 );
    DecimalFormat precision2 = new DecimalFormat( "0.00" );

    // get coordinates and radius
    String output = "X coordinate is " + circle.getX() +
        "\nY coordinate is " + circle.getY() +
        "\nRadius is " + circle.getRadius();

    // set coordinates and radius
    circle.setRadius( 4.25 );
    circle.setPoint( 2, 2 );

    // get String representation of Circle and calculate area
    output +=
        "\nThe new location and radius of c are\n" + circle +
        "\nArea is " + precision2.format( circle.area() );

    JOptionPane.showMessageDialog( null, output,
        "Demonstrating Class Circle",
        JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
} // end class Test

Fig. 9.13 Testing class Circle (part 2 of 2).
Our last example is shown in Fig. 9.14 and Fig. 9.15. Figure 9.14 shows the Cylinder class definition with the Cylinder method definitions. Note that class Cylinder extends class Circle. This means that the public interface to Cylinder includes the Circle methods and Point methods as well as the Cylinder constructor and Cylinder methods setHeight, getHeight, area (which overrides the Circle area method), volume and toString.

```java
// Fig. 9.14: Cylinder.java
// Definition of class Cylinder
package com.deitel.jhtp4.ch09;

public class Cylinder extends Circle {
    protected double height; // height of Cylinder

    // no-argument constructor
    public Cylinder() {
        // implicit call to superclass constructor here
        setHeight(0);
    }

    // constructor
    public Cylinder(double cylinderHeight, double cylinderRadius, int xCoordinate, int yCoordinate) {
        // call superclass constructor to set coordinates/radius
        super(cylinderRadius, xCoordinate, yCoordinate);

        // set cylinder height
        setHeight(cylinderHeight);
    }

    // set height of Cylinder
    public void setHeight(double cylinderHeight) {
        height = (cylinderHeight >= 0 ? cylinderHeight : 0);
    }

    // get height of Cylinder
    public double getHeight() {
        return height;
    }

    // calculate area of Cylinder (i.e., surface area)
    public double area() {
        return 2 * super.area() +
            2 * Math.PI * radius * height;
    }
}
```

Fig. 9.14 Class Cylinder definition (part 1 of 2).
Method `main` of the Test application (Fig. 9.15) instantiates an object of class `Cylinder` (line 19), then uses `get` methods (lines 23–26) to obtain information about the `Cylinder` object. Again, the Test applications’s `main` method cannot reference directly the protected data of class `Cylinder`. Method `main` uses `set` methods `setHeight`, `setRadius` and `setPoint` (lines 29–31) to reset the `height`, `radius` and coordinates of the `Cylinder`. Then `main` uses `toString`, `area` and `volume` to print the attributes and some facts about the `Cylinder`. Figure 9.15 is a Test application to test class `Cylinder`’s capabilities.
The series of examples in this section nicely demonstrates inheritance and defining and referencing protected instance variables. The reader should now be confident with the basics of inheritance. In the next several sections, we show how to program with inheritance hierarchies in a general manner, using polymorphism. Data abstraction, inheritance and polymorphism are the crux of object-oriented programming.

### 9.10 Introduction to Polymorphism

With polymorphism, it is possible to design and implement systems that are more easily extensible. Programs can be written to process generically—as superclass objects—objects of all existing classes in a hierarchy. Classes that do not exist during program development can be added with little or no modifications to the generic part of the program—as long as those classes are part of the hierarchy that is being processed generically. The only parts of...
a program that need modification are those parts that require direct knowledge of the particular class that is added to the hierarchy. We will study two substantial class hierarchies and will show how objects throughout those hierarchies are manipulated polymorphically.

### 9.11 Type Fields and switch Statements

One means of dealing with objects of many different types is to use a `switch` statement to take an appropriate action on each object, based on that object’s type. For example, in a hierarchy of shapes in which each shape has a `shapeType` instance variable, a `switch` structure could determine which `print` method to call based on the object’s `shapeType`.

There are many problems with using `switch` logic. The programmer might forget to make such a type test when one is warranted. The programmer might forget to test all possible cases in a `switch`. If a `switch`-based system is modified by adding new types, the programmer might forget to insert the new cases in existing `switch` statements. Every addition or deletion of a class demands that every `switch` statement in the system be modified; tracking these down can be time consuming and error prone.

As we will see, polymorphic programming can eliminate the need for `switch` logic. The programmer can use Java’s polymorphism mechanism to perform the equivalent logic automatically, thus avoiding the kinds of errors typically associated with `switch` logic.

**Testing and Debugging Tip 9.2**

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic in favor of simpler sequential code. This simplification facilitates testing, debugging, and program maintenance.

### 9.12 Dynamic Method Binding

Suppose a set of shape classes such as `Circle`, `Triangle`, `Rectangle`, `Square`, are all derived from superclass `Shape`. In object-oriented programming, each of these classes might be endowed with the ability to draw itself. Each class has its own `draw` method, and the `draw` method implementation for each shape is quite different. When drawing a shape, whatever that shape may be, it would be nice to be able to treat all these shapes generically as objects of the superclass `Shape`. Then, to draw any shape, we could simply call method `draw` of superclass `Shape` and let the program determine dynamically (i.e., at execution time) which subclass `draw` method to use from the actual object’s type.

To enable this kind of behavior, we declare `draw` in the superclass, and then we override `draw` in each of the subclasses to draw the appropriate shape.

**Software Engineering Observation 9.16**

When a subclass chooses not to redefine a method, the subclass simply inherits its immediate superclass’s method definition.

If we use a superclass reference to refer to a subclass object and invoke the `draw` method, the program will choose the correct subclass’s `draw` method dynamically (i.e., at execution time). This is called dynamic method binding. Dynamic method binding is an important mechanism for implementing polymorphic processing of objects and will be illustrated in the case studies later in this chapter.
9.13 final Methods and Classes

We saw in Chapter 7 that variables can be declared final to indicate that they cannot be modified after they are declared and that they must be initialized when they are declared. It is also possible to define methods and classes with the final modifier.

A method that is declared final cannot be overridden in a subclass. Methods that are declared static and methods that are declared private are implicitly final. Because a final method’s definition can never change, the compiler can optimize the program by removing calls to final methods and replacing them with the expanded code of their definitions at each method call location—a technique known as inlining the code.

A class that is declared final cannot be a superclass (i.e., a class cannot inherit from a final class). All methods in a final class are implicitly final.

Performance Tip 9.2

The compiler can decide to inline a final method call and will do so for small, simple final methods. Inlining does not violate encapsulation or information hiding (but does improve performance because it eliminates the overhead of making a method call).

Software Engineering Observation 9.17

A class declared final cannot be extended, and every method in it is implicitly final.

Software Engineering Observation 9.18

In the Java API, the vast majority of the thousands of classes are not declared final. This enables inheritance and polymorphic processing—the fundamental capabilities of object-oriented programming. However, in some cases it is important to declare classes final—typically for security or performance reasons.

9.14 Abstract Superclasses and Concrete Classes

When we think of a class as a type, we assume that objects of that type will be instantiated. However, there are cases in which it is useful to define classes for which the programmer never intends to instantiate any objects. Such classes are called abstract classes. Because these are used as superclasses in inheritance situations, we will normally refer to them as abstract superclasses. No objects of abstract superclasses can be instantiated.

The sole purpose of an abstract class is to provide an appropriate superclass from which other classes may inherit interface and/or implementation (we will see examples of each shortly). Classes from which objects can be instantiated are called concrete classes.

1. Class String is an example of a final class. This class cannot be extended, so programs that use Strings can rely on the functionality of String objects as specified in the Java API. Making the class final also prevents programmers from creating subclasses that might bypass security restrictions. For example, when a Java program attempts to open a file on your computer, the program supplies a String representing the name of the file. In many cases, opening a file is subject to security restrictions. If it were possible to create a subclass of String, that subclass might be implemented in a manner that enables it to specify one String to pass a security or permissions test, then specify a different name when the program actually opens the file. For more information on final classes and methods visit: [java.sun.com/docs/books/tutorial/java/javaOO/final.html](http://java.sun.com/docs/books/tutorial/java/javaOO/final.html) [Note: Strings are covered in detail in Chapter 10 and file processing is covered in detail in Chapter 16.]
We could have an abstract superclass `TwoDimensionalObject` and derive concrete classes, such as `Square`, `Circle`, `Triangle`. We could also have an abstract superclass `ThreeDimensionalObject` and derive such concrete classes as `Cube`, `Sphere` and `Cylinder`. Abstract superclasses are too generic to define real objects; we need to be more specific before we can think of instantiating objects. For example, if someone tells you to “draw the shape,” what shape would you draw? Concrete classes provide the specifics that make it reasonable to instantiate objects.

A class is made abstract by declaring it with keyword `abstract`. A hierarchy does not need to contain any `abstract` classes, but, as we will see, many good object-oriented systems have class hierarchies headed by `abstract` superclasses. In some cases, `abstract` classes constitute the top few levels of the hierarchy. A good example of this is the shape hierarchy in Fig. 9.3. The hierarchy begins with `abstract` superclass `Shape`. On the next level down we have two more `abstract` superclasses, namely `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level down would start defining concrete classes for such two-dimensional shapes as `Circle` and `Square` and for such three-dimensional shapes as `Sphere` and `Cube`.

### 9.15 Polymorphism Examples

Here is an example of polymorphism. If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object. An operation (such as calculating the perimeter or the area) that can be performed on an object of class `Quadrilateral` can also be performed on an object of class `Rectangle`. Such operations can also be performed on other “kinds of” `quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`. When a request is made through a superclass reference to use a method, Java chooses the correct overridden method polymorphically in the appropriate subclass associated with the object.

Here is another example of polymorphism. Suppose we have a video game that manipulates objects of many varieties, including objects of class `Martian`, `Venutian`, `Plutonian`, `SpaceShip`, `LaserBeam` and the like. Each of these classes extends a common superclass, `GamePiece`, that contains a method called `drawYourself`. This method is defined by each subclass. A Java screen manager program would simply maintain some kind of collection (such as a `GamePiece` array) of references to objects of these various classes. To refresh the screen periodically, the screen manager would simply send each object the same message, namely `drawYourself`. Each object would respond in its own unique way. A `Martian` object would draw itself with the appropriate number of antennae. A `SpaceShip` object would draw itself bright and silvery. A `LaserBeam` object would draw itself as a bright red beam across the screen. Thus, the same message sent to a variety of objects would have “many forms” of results—hence the term *polymorphism*.

Such a polymorphic screen manager makes it especially easy to add new types of objects to a system with minimal impact. Suppose we want to add `Mercurians` to our video game. We certainly have to build a new class `Mercurian` that extends `GamePiece` and provides its own definition of the `drawYourself` method. Then, when objects of class `Mercurian` appear in the collection, the screen manager need not be modified. It simply sends the message `drawYourself` to every object in the collection.
regardless of the object’s type, so the new Mercurian objects just “fit right in.” Thus, with polymorphism, new types of objects not even envisioned when a system is created can be added without modifications to the system (other than the new class itself, of course).

Through the use of polymorphism, one method call can cause different actions to occur, depending on the type of the object receiving the call. This gives the programmer tremendous expressive capability. We will see examples of the power of polymorphism in the next several sections.

**Software Engineering Observation 9.19**

With polymorphism, the programmer can deal in generalities and let the execution-time environment concern itself with the specifics. The programmer can command a wide variety of objects to behave in manners appropriate to those objects without even knowing the types of those objects.

**Software Engineering Observation 9.20**

Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independent of the types of the objects to which messages (i.e., method calls) are sent. Thus, new types of objects that can respond to existing messages can be added into such a system without modifying the base system.

**Software Engineering Observation 9.21**

If a method is declared final, it cannot be overridden in subclasses, so method calls may not be sent polymorphically to objects of those subclasses. The method call may still be sent to subclasses, but they will all respond identically rather than polymorphically.

**Software Engineering Observation 9.22**

An abstract class defines a common interface for the various members of a class hierarchy. The abstract class contains methods that will be defined in the subclasses. All classes in the hierarchy can use this same interface through polymorphism.

Although we cannot instantiate objects of abstract superclasses, we can declare references to abstract superclasses. Such references can be used to enable polymorphic manipulations of subclass objects when such objects are instantiated from concrete classes.

Let us consider more applications of polymorphism. A screen manager needs to display a variety of objects, including new types of objects that will be added to the system even after the screen manager is written. The system may need to display various shapes (the superclass is Shape), such as Circle, Triangle and Rectangle. Each shape class is derived from superclass Shape. The screen manager uses superclass Shape references to manage the objects to be displayed. To draw any object (regardless of the level at which that object appears in the inheritance hierarchy), the screen manager uses a superclass reference to the object and simply sends a draw message to the object. Method draw has been declared abstract in superclass Shape and has been overridden in each of the subclasses. Each Shape object knows how to draw itself. The screen manager does not have to worry about what type each object is or whether the screen manager has seen objects of that type before—the screen manager simply tells each object to draw itself.

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices can have a
certain uniformity. The *write* message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the *write* call itself is really no different from the *write* to any other device in the system—simply place some number of bytes from memory onto that device. An object-oriented operating system might use an *abstract* superclass to provide an interface appropriate for all device drivers. Then, through inheritance from that *abstract* superclass, subclasses are formed that all operate similarly. The capabilities (i.e., the *public* interface) offered by the device drivers are provided as *abstract* methods in the *abstract* superclass. The implementations of these *abstract* methods are provided in the subclasses that correspond to the specific types of device drivers.

It is common in object-oriented programming to define an *iterator class* that can walk through all the objects in a collection (such as an array). If you want to print a list of objects in a linked list, for example, an iterator object can be instantiated that will return the next element of the linked list each time the iterator is called. Iterators are commonly used in polymorphic programming to walk through an array or a linked list of objects from various levels of a hierarchy. The references in such a list would all be superclass references (see Chapter 19, Data Structures, for more on linked lists). A list of objects of superclass class *TwoDimensionalShape* could contain objects from the classes *Square*, *Circle*, *Triangle* and so on. Sending a *draw* message to each object in the list would, using polymorphism, draw the correct picture on the screen.

### 9.16 Case Study: A Payroll System Using Polymorphism

Let us use *abstract* classes, *abstract* methods and polymorphism to perform payroll calculations based on the type of employee (Fig. 9.16). We use an *abstract* superclass *Employee*. The subclasses of *Employee* are *Boss* (Fig. 9.17)—paid a fixed weekly salary regardless of the number of hours worked, *CommissionWorker* (Fig. 9.18)—paid a flat base salary plus a percentage of sales, *PieceWorker* (Fig. 9.19)—paid by the number of items produced and *HourlyWorker* (Fig. 9.20)—paid by the hour and receives overtime pay. Each subclass of *Employee* has been declared *final*, because we do not intend to inherit from them again.

An *earnings* method call certainly applies generically to all employees. But the way each person’s earnings are calculated depends on the class of the employee, and these classes are all derived from the superclass *Employee*. So *earnings* is declared *abstract* in superclass *Employee* and appropriate implementations of *earnings* are provided for each of the subclasses. Then, to calculate any employee’s earnings, the program simply uses a superclass reference to that employee’s object and invokes the *earnings* method. In a real payroll system, the various *Employee* objects might be referenced by individual elements in an array of *Employee* references. The program would simply walk through the array one element at a time, using the *Employee* references to invoke the *earnings* method of each object.

**Software Engineering Observation 9.23**

If a subclass is derived from a superclass with an *abstract* method, and if no definition is supplied in the subclass for that *abstract* method (i.e., if that method is not overridden in the subclass), that method remains *abstract* in the subclass. Consequently, the subclass is also an *abstract* class and must be explicitly declared as an *abstract* class.
Software Engineering Observation 9.24

The ability to declare an abstract method gives the class designer considerable power over how subclasses will be implemented in a class hierarchy. Any new class that wants to inherit from this class is forced to override the abstract method (either directly or by inheriting from a class that has overridden the method). Otherwise, that new class will contain an abstract method and thus be an abstract class, unable to instantiate objects.

Software Engineering Observation 9.25

An abstract can still have instance data and nonabstract methods subject to the normal rules of inheritance by subclasses. An abstract class can also have constructors.

Common Programming Error 9.7

Attempting to instantiate an object of an abstract class (i.e., a class that contains one or more abstract methods) is a syntax error.

Common Programming Error 9.8

It is a syntax error if a class with one or more abstract methods is not explicitly declared abstract.

Let us consider the Employee class (Fig. 9.16). The public methods include a constructor that takes the first name and last name as arguments; a getFirstName method that returns the first name; a getLastName method that returns the last name; a toString method that returns the first name and last name separated by a space; and an abstract method—earnings. Why is this method abstract? The answer is that it does not make sense to provide an implementation of this method in the Employee class. We cannot calculate the earnings for a generic employee—we must first know what kind of employee it is. By making this method abstract we are indicating that we will provide an implementation in each concrete subclass, but not in the superclass itself.

```java
public abstract class Employee {
    private String firstName;
    private String lastName;

    // constructor
    public Employee( String first, String last )
    {
        firstName = first;
        lastName = last;
    }

    // get first name
    public String getFirstName()
    {
        return firstName;
    }
```

Fig. 9.16 Employee abstract superclass (part 1 of 2).
Class **Boss** (Fig. 9.17) is derived from **Employee**. The **public** methods include a constructor that takes a first name, a last name and a weekly salary as arguments and passes the first name and last name to the **Employee** constructor to initialize the **firstName** and **lastName** members of the superclass part of the subclass object. Other **public** methods include a **setWeeklySalary** method to assign a new value to **private** instance variable **weeklySalary**; an **earnings** method defining how to calculate a **Boss**’s earnings; and a **toString** method that forms a **String** containing the type of the employee (i.e., "**Boss:** ") followed by the boss’s name.
Class CommissionWorker (Fig. 9.18) is derived from Employee. The public methods include a constructor that takes a first name, a last name, a salary, a commission and a quantity of items sold as arguments and passes the first name and last name to the Employee constructor; set methods to assign new values to instance variables salary, commission and quantity; an earnings method to calculate a CommissionWorker’s earnings; and a toString method that forms a String containing the employee type (i.e., "Commissionworker: ") followed by the worker’s name.

```java
// Fig. 9.18: CommissionWorker.java
// CommissionWorker class derived from Employee

public final class CommissionWorker extends Employee {
    private double salary;      // base salary per week
    private double commission;  // amount per item sold
    private int quantity;       // total items sold for week

    // constructor for class CommissionWorker
    public CommissionWorker( String first, String last,
                              double salary, double commission, int quantity )
    {
        super( first, last );  // call superclass constructor
        setSalary( salary );
        setCommission( commission );
        setQuantity( quantity );
    }

    // set CommissionWorker's weekly base salary
    public void setSalary( double weeklySalary )
    {
        salary = ( weeklySalary > 0 ? weeklySalary : 0 );
    }

    // get Boss's pay
    public double earnings()
    {
        return weeklySalary;
    }

    // get String representation of Boss's name
    public String toString()
    {
        return "Boss: " + super.toString();
    }
}
```
Fig. 9.18  **CommissionWorker** extends **abstract** class **Employee** (part 2 of 2).

Class **PieceWorker** (Fig. 9.19) is derived from **Employee**. The **public** methods include a constructor that takes a first name, a last name, a wage per piece and a quantity of items produced as arguments and passes the first name and last name to the **Employee** constructor; **set** methods to assign new values to instance variables **wagePerPiece** and **quantity**; an **earnings** method defining how to calculate a **PieceWorker**'s earnings; and a **toString** method that forms a **String** containing the type of the employee (i.e., "**Piece worker:**") followed by the pieceworker’s name.

Fig. 9.19  **PieceWorker** extends **abstract** class **Employee** (part 1 of 2).
Fig. 9.20

Object-Oriented Programming
Chapter 9

16  // set PieceWorker's wage
17  public void setWage( double wage )
18  {
19      wagePerPiece = ( wage > 0 ? wage : 0 );
20  }
21
22  // set number of items output
23  public void setQuantity( int numberOfItems )
24  {
25      quantity = ( numberOfItems > 0 ? numberOfItems : 0 );
26  }
27
28  // determine PieceWorker's earnings
29  public double earnings()
30  {
31      return quantity * wagePerPiece;
32  }
33
34  public String toString()
35  {
36      return "Piece worker: " + super.toString();
37  }
38
39  }
40  // end class PieceWorker

Fig. 9.19  PieceWorker extends abstract class Employee (part 2 of 2).

Class HourlyWorker (Fig. 9.20) is derived from Employee. The public
methods include a constructor that takes a first name, a last name, a wage and the number
of hours worked as arguments and passes the first name and last name to the Employee
constructor; set methods to assign new values to instance variables wage and hours; an
earnings method defining how to calculate an HourlyWorker’s earnings; and a
toString method that forms a String containing the type of the employee (i.e.,
"Hourly worker: ") followed by the hourly worker’s name.

Fig. 9.20  HourlyWorker extends abstract class Employee (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
Method `main` of the `Test` application (Fig. 9.21) begins by declaring `Employee` reference, `ref`. Each of the types of `Employees` is handled similarly in `main`, so we will discuss only the case in which `main` deals with a `Boss` object.

```java
// Fig. 9.21: Test.java
// Driver for Employee hierarchy

import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class Test {
    // test Employee hierarchy
    public static void main(String args[]) {
        Employee employee; // superclass reference
        String output = "";
        Boss boss = new Boss("John", "Smith", 800.0);
    }
}
```

Fig. 9.21 Testing the `Employee` class hierarchy using an `abstract` superclass.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
CommissionWorker commissionWorker =
    new CommissionWorker(
        "Sue", "Jones", 400.0, 3.0, 150 );

PieceWorker pieceWorker =
    new PieceWorker( "Bob", "Lewis", 2.5, 200 );

HourlyWorker hourlyWorker =
    new HourlyWorker( "Karen", "Price", 13.75, 40 );

DecimalFormat precision2 = new DecimalFormat( "0.00" );

// Employee reference to a Boss
employee = boss;

output += employee.toString() + " earned $" +
    precision2.format( employee.earnings() ) + "\n" +
    boss.toString() + " earned $" +
    precision2.format( boss.earnings() ) + "\n";

// Employee reference to a CommissionWorker
employee = commissionWorker;

output += employee.toString() + " earned $" +
    precision2.format( employee.earnings() ) + "\n" +
    commissionWorker.toString() + " earned $" +
    precision2.format( commissionWorker.earnings() ) + "\n";

// Employee reference to a PieceWorker
employee = pieceWorker;

output += employee.toString() + " earned $" +
    precision2.format( employee.earnings() ) + "\n" +
    pieceWorker.toString() + " earned $" +
    precision2.format( pieceWorker.earnings() ) + "\n";

// Employee reference to an HourlyWorker
employee = hourlyWorker;

output += employee.toString() + " earned $" +
    precision2.format( employee.earnings() ) + "\n" +
    hourlyWorker.toString() + " earned $" +
    precision2.format( hourlyWorker.earnings() ) + "\n";

JOptionPane.showMessageDialog( null, output,
    "Demonstrating Polymorphism",
    JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 );

} // end class Test

Fig. 9.21 Testing the Employee class hierarchy using an abstract superclass.
Fig. 9.21 Testing the **Employee** class hierarchy using an **abstract** superclass.

Line 18 instantiates subclass object class **Boss** and provides various constructor arguments, including a first name, a last name and a fixed weekly salary. The new object is assigned to **Boss** reference **boss**.

Line 33 assigns to superclass **Employee** reference **employee** a reference to the subclass **Boss** object to which **boss** refers. This is precisely what we must do to effect polymorphic behavior.

The method call in line 35 invokes the **toString** method of the object referenced by **employee**. The system determines that the referenced object is a **Boss** and invokes subclass **Boss**’s **toString** method—again, polymorphic behavior. This method call is an example of dynamic method binding—the decision as to what method to invoke is deferred until execution time.

The method call in line 36 invokes the **earnings** method of the object to which **employee** refers. The system determines that the object is a **Boss** and invokes the subclass **Boss**’s **earnings** method rather than the superclass’s **earnings** method. This is also an example of dynamic method binding.

The method call in line 37 explicitly invokes the **Boss** version of method **toString** by following reference **boss** with the dot operator. We included line 37 for comparison purposes to ensure that the dynamically bound method invoked with **employee.toString()** was indeed the proper method.

The method call in line 38 explicitly invokes the **Boss** version of method **earnings** by using the dot operator with the specific **Boss** reference **boss**. This call is also included for comparison purposes to ensure that the dynamically bound method invoked with **employee.earnings()** was indeed the proper method.

To prove that the superclass reference **employee** can be used to invoke **toString** and **earnings** for the other types of employees, lines 41, 50 and 58 each assign a different type of employee object (**CommissionWorker**, **PieceWorker** and **HourlyWorker**, respectively) to the superclass reference **employee**; then the two methods are called after each assignment to show that Java is always capable of determining the type of the referenced object before invoking a method.

### 9.17 New Classes and Dynamic Binding

Polymorphism certainly works nicely when all possible classes are known in advance. But it also works when new kinds of classes are added to systems.
New classes are accommodated by dynamic method binding (also called late binding). An object’s type need not be known at compile time for a polymorphic call to be compiled. At execution time, the call is matched with the method of the called object.

A screen manager program can now handle (without recompilation) new types of display objects as they are added to the system. The `draw` method call remains the same. The new objects themselves each contain a `draw` method implementing the actual drawing capabilities. This makes it easy to add new capabilities to systems with minimal impact. It also promotes software reuse.

**Performance Tip 9.3**

The kinds of polymorphic manipulations made possible with dynamic binding can also be accomplished by using hand-coded `switch` logic based on type fields in objects. The polymorphic code generated by the Java compiler runs with performance comparable to that of efficiently coded `switch` logic.

**Software Engineering Observation 9.26**

Java provides mechanisms for loading classes into a program dynamically to enhance the functionality of an executing program. In particular, static method `forName` of class `Class` (package `java.lang`) can be used to load a class definition, then create new objects of that class for use in a program. This concept is beyond the scope of this book. For more information, see the online API documentation for `Class`.

### 9.18 Case Study: Inheriting Interface and Implementation

Our next example (Fig. 9.22–Fig. 9.26) reexamines the `Point`, `Circle`, `Cylinder` hierarchy, except that we now head the hierarchy with abstract superclass `Shape` (Fig. 9.22). This hierarchy mechanically demonstrates the power of polymorphism. In the exercises, we explore a more realistic hierarchy of shapes.

`Shape` contains abstract method `getName`, so `Shape` must be declared an abstract superclass. `Shape` contains two other methods, `area` and `volume`, each of which has an implementation that returns zero by default. `Point` inherits these implementations from `Shape`. This makes sense because both the area and volume of a point are zero. `Circle` inherits the `volume` method from `Point`, but `Circle` provides its own implementation for the `area` method. `Cylinder` provides its own implementations for both the `area` (interpreted as the surface area of the cylinder) and `volume` methods.

In this example, class `Shape` is used to define a set of methods that all `Shapes` in our hierarchy have in common. Defining these methods in class `Shape` enables us to generically call these methods through a `Shape` reference. Remember, the only methods that can be called through any reference are those public methods defined in the reference’s declared class type and any public methods inherited into that class. Thus, we can call `Object` and `Shape` methods through a `Shape` reference.

Note that although `Shape` is an abstract superclass, it still contains implementations of methods `area` and `volume`, and these implementations are inheritable. The `Shape` class provides an inheritable interface (set of services) in the form of three methods that all classes of the hierarchy will contain. The `Shape` class also provides some implementations that subclasses in the first few levels of the hierarchy will use.

This case study emphasizes that a subclass can inherit interface and/or implementation from a superclass.
Software Engineering Observation 9.27
Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy—each new subclass inherits one or more methods that were defined in a superclass and uses the superclass definitions.

Software Engineering Observation 9.28
Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy—a superclass specifies one or more methods that should be called identically for each object in the hierarchy (i.e., they have the same signature), but the individual subclasses provide their own implementations of the method(s).

Superclass Shape (Fig. 9.22) extends Object, consists of three public methods and does not contain any data (although it could). Method getName is abstract, so it is overridden in each of the subclasses. Methods area and volume are defined to return 0.0. These methods are overridden in subclasses when it is appropriate for those classes to have a different area calculation (classes Circle and Cylinder) and/or a different volume calculation (class Cylinder).

Class Point (Fig. 9.23) is derived from Shape. A Point has an area of 0.0 and a volume of 0.0, so the superclass methods area and volume are not overridden here—they are inherited as defined in Shape. Other methods include setPoint to assign new x and y coordinates to a Point and getX and getY to return the x and y coordinates of a Point. Method getName is an implementation of the abstract method in the superclass. If this method were not defined, class Point would be an abstract class.

```java
// Fig. 9.22: Shape.java
// Definition of abstract base class Shape

public abstract class Shape extends Object {
   // return shape's area
   public double area() {
      return 0.0;
   }

   // return shape's volume
   public double volume() {
      return 0.0;
   }

   // abstract method must be defined by concrete subclasses
   // to return appropriate shape name
   public abstract String getName();
}
```

Fig. 9.22 Shape abstract superclass for Point, Circle, Cylinder hierarchy.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
public class Point extends Shape {
    protected int x, y; // coordinates of the Point

    // no-argument constructor
    public Point()
    {
        setPoint( 0, 0 );
    }

    // constructor
    public Point( int xCoordinate, int yCoordinate )
    {
        setPoint( xCoordinate, yCoordinate );
    }

    // set x and y coordinates of Point
    public void setPoint( int xCoordinate, int yCoordinate )
    {
        x = xCoordinate;
        y = yCoordinate;
    }

    // get x coordinate
    public int getX()
    {
        return x;
    }

    // get y coordinate
    public int getY()
    {
        return y;
    }

    // convert point into String representation
    public String toString()
    {
        return "[" + x + ", " + y + "]";
    }

    // return shape name
    public String getName()
    {
        return "Point";
    }
} // end class Point

Fig. 9.23 Point subclass of abstract class Shape.
Class **Circle** (Fig. 9.24) is derived from **Point**. A **Circle** has a volume of 0.0, so superclass method **volume** is not overridden—it is inherited from class **Point**, which inherited it from **Shape**. A **Circle** has an area different from that of a **Point**, so the **area** method is overridden. Method **getName** is an implementation of the **abstract** method in the superclass. If this method is not overridden here, the **Point** version of **getName** would be inherited. Other methods include **setRadius** to assign a new radius to a **Circle** and **getRadius** to return the radius of a **Circle**.

```java
// Fig. 9.24: Circle.java
// Definition of class Circle

public class Circle extends Point {  // inherits from Point
    protected double radius;

    // no-argument constructor
    public Circle() {
        // implicit call to superclass constructor here
        setRadius( 0 );
    }

    // constructor
    public Circle( double circleRadius, int xCoordinate,
                   int yCoordinate ) {
        // call superclass constructor
        super( xCoordinate, yCoordinate );
        setRadius( circleRadius );
    }

    // set radius of Circle
    public void setRadius( double circleRadius ) {
        radius = ( circleRadius >= 0 ? circleRadius : 0 );
    }

    // get radius of Circle
    public double getRadius() {
        return radius;
    }

    // calculate area of Circle
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

**Fig. 9.24**: **Circle** subclass of **Point**—indirect subclass of **abstract** class **Shape** (part 1 of 2).
Software Engineering Observation 9.29

A subclass always inherits the most recently defined version of each public and protected method from its direct and indirect superclasses.

Class Cylinder (Fig. 9.25) is derived from Circle. A Cylinder has area and volume different from those of class Circle, so the area and volume methods are both overridden. Method getName is an implementation of the abstract method in the superclass. If this method had not been overridden here, the Circle version of getName would be inherited. Other methods include setHeight to assign a new height to a Cylinder and getHeight to return the height of a Cylinder.

```java
// Fig. 9.25: Cylinder.java
// Definition of class Cylinder.

public class Cylinder extends Circle {
    protected double height; // height of Cylinder

    // no-argument constructor
    public Cylinder() {
        // implicit call to superclass constructor here
        setHeight( 0 );
    }

    // constructor
    public Cylinder( double cylinderHeight, double cylinderRadius, int xCoordinate, int yCoordinate )
    {
        // call superclass constructor
        super( cylinderRadius, xCoordinate, yCoordinate );
    }
}
```

Fig. 9.25 Cylinder subclass of Circle—indirect subclass of abstract class Shape (part 1 of 2).
Method `main` of class `Test` (Fig. 9.26) instantiates `Point` object `point`, `Circle` object `circle` and `Cylinder` object `cylinder` (lines 16–18). Next, array `arrayOfShapes` is instantiated (line 21). This array of superclass `Shape` references will refer to each subclass object instantiated. At line 24, the reference `point` is assigned to array element `arrayOfShapes[0]`. At line 27, the reference `circle` is assigned to array element `arrayOfShapes[1]`. At line 30, the reference `cylinder` is assigned to array element `arrayOfShapes[2]`. Now, each superclass `Shape` reference in the array refers to a subclass object of type `Point`, `Circle` or `Cylinder`.

**Fig. 9.25** Cylinder subclass of Circle—indirect subclass of abstract class Shape (part 2 of 2).
// Fig. 9.26: Test.java
// Class to test Shape, Point, Circle, Cylinder hierarchy

// Java core packages
import java.text.DecimalFormat;

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

public class Test {

    // test Shape hierarchy
    public static void main( String args[] ) {
        // create shapes
        Point point = new Point( 7, 11 );
        Circle circle = new Circle( 3.5, 22, 8 );
        Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );

        // create Shape array
        Shape arrayOfShapes[] = new Shape[3];

        // aim arrayOfShapes[0] at subclass Point object
        arrayOfShapes[0] = point;

        // aim arrayOfShapes[1] at subclass Circle object
        arrayOfShapes[1] = circle;

        // aim arrayOfShapes[2] at subclass Cylinder object

        // get name and String representation of each shape
        String output =
            point.getName() + "\n" + point.toString() + "\n" +
            circle.getName() + "\n" + circle.toString() + "\n" +
            cylinder.getName() + "\n" + cylinder.toString();

        DecimalFormat precision2 = new DecimalFormat( "0.00" );

        // loop through arrayOfShapes and get name, area and volume of each shape in arrayOfShapes
        for ( int i = 0; i < arrayOfShapes.length; i++ ) {
            output += "\n" + arrayOfShapes[i].getName() + "\n" +
                arrayOfShapes[i].toString() + "\n" +
                precision2.format(arrayOfShapes[i].area()) + "\n" +
                precision2.format(arrayOfShapes[i].volume() );
        }

        JOptionPane.showMessageDialog( null, output,
            "Demonstrating Polymorphism",
            JOptionPane.INFORMATION_MESSAGE );
    }

} // End class Test

Fig. 9.26  Shape, Point, Circle, Cylinder hierarchy (part 1 of 2).
Lines 33 through 36 invoke methods `getName` and `toString` to illustrate that the objects are initialized correctly (as in the first three lines of the screen capture).

Next, the `for` structure at lines 42–49 walks through `arrayOfShapes`, and the following calls are made during each iteration of the loop:

```java
arrayOfShapes[i].getName()
arrayOfShapes[i].toString()
arrayOfShapes[i].area()
arrayOfShapes[i].volume()
```

Each of these method calls is invoked on the object to which `arrayOfShapes[i]` currently refers. When the compiler looks at each of these calls, it is simply trying to determine whether a `Shape` reference (`arrayOfShapes[i]`) can be used to call these methods. For methods `getName`, `area` and `volume` the answer is yes, because each of these methods is defined in class `Shape`. For method `toString`, the compiler first looks at class `Shape` to determine that `toString` is not defined there, then the compiler proceeds to `Shape`’s superclass (`Object`) to determine whether `Shape` inherited a `toString` method that takes no arguments (as it did, because all `Objects` have a `toString` method).

The screen capture illustrates that all four methods are invoked properly based on the type of the referenced object. First, the string "Point: " and the coordinates of the object `point` (`arrayOfShapes[0]`) are output; the area and volume are both 0. Next, the string "Circle: ", the coordinates of object `circle`, and the radius of object `circle` (`arrayOfShapes[1]`) are output; the area of `circle` is calculated and the volume is...
Finally, the string "Cylinder: ", the coordinates of object \texttt{cylinder}, the radius of object \texttt{cylinder} and the height of object \texttt{cylinder} (arrayOfShapes[2]) are output; the area of \texttt{cylinder} is calculated and the volume of \texttt{cylinder} is calculated. All the method calls to \texttt{getName}, \texttt{toString}, \texttt{area} and \texttt{volume} are resolved at run-time with dynamic binding.

\section*{9.19 Case Study: Creating and Using Interfaces}

Our next example (Fig. 9.27–Fig. 9.31) reexamines the \texttt{Point}, \texttt{Circle}, \texttt{Cylinder} hierarchy one last time, replacing \texttt{abstract} superclass \texttt{Shape} with the interface \texttt{Shape} (Fig. 9.27). An interface definition begins with the keyword \texttt{interface} and contains a set of \texttt{public abstract} methods. Interfaces may also contain \texttt{public static final} data. To use an interface, a class must specify that it \texttt{implements} the interface and the class must define every method in the interface with the number of arguments and the return type specified in the interface definition. If the class leaves any method of the interface undefined, the class becomes an \texttt{abstract} class and must be declared \texttt{abstract} in the first line of its class definition. Implementing a interface is like signing a contract with the compiler that states, “I will define all the methods specified by the interface.”

\begin{quote}
\textbf{Common Programming Error 9.9} \\
Leaving a method of an \texttt{interface} undefined in a class that \texttt{implements} the interface results in a compile error indicating that the class must be declared \texttt{abstract}.
\end{quote}

\begin{quote}
\textbf{Software Engineering Observation 9.30} \\
Declaring a \texttt{final} reference indicates that the reference always refers to the same object. However, this does not affect the object to which the reference refers—the object’s data still can be modified.
\end{quote}

We started using the concept of an interface when we introduced GUI event handling in Chapter 6, “Methods.” Recall that our applet class included \texttt{implements ActionListener} (an interface in package \texttt{java.awt.event}). The reason we were required to define \texttt{actionPerformed} in the applets with event handling is that \texttt{ActionListener} is an interface that specifies that \texttt{actionPerformed} must be defined. Interfaces are an important part of GUI event handling, as we will discuss in the next section.

An interface is typically used in place of an \texttt{abstract} class when there is no default implementation to inherit—i.e., no instance variables and no default method implementations. Like \texttt{public abstract} classes, \texttt{interfaces} are typically \texttt{public} data types, so they are normally defined in files with the same name as the interface and the .\texttt{java} extension.

The definition of interface \texttt{Shape} begins in Fig. 9.27 at line 4. Interface \texttt{Shape} has \texttt{abstract} methods \texttt{area}, \texttt{volume} and \texttt{getName}. By coincidence, all three methods take no arguments. However, this is not a requirement of methods in an interface.

In Fig. 9.28, line 4 indicates that class \texttt{Point} extends class \texttt{Object} and implements interface \texttt{Shape}. Class \texttt{Point} provides definitions of all three methods in the interface. Method \texttt{area} is defined at lines 45–48. Method \texttt{volume} is defined at lines 51–54. Method \texttt{getName} is defined at lines 57–60. These three methods satisfy the implementation requirement for the three methods defined in the interface. We have fulfilled our contract with the compiler.
Fig. 9.27  Point, circle, cylinder hierarchy with a Shape interface.

Fig. 9.28  Point implementation of interface Shape (part 1 of 2).
When a class implements an interface, the same is a relationship provided by inheritance applies. For example, class Point implements Shape. Therefore, a Point object is a Shape. In fact, objects of any class that extends Point are also Shape objects. Using this relationship, we have maintained the original definitions of class Circle, class Cylinder and application class Test from Section 9.18 (repeated in Fig. 9.29–Fig. 9.31) to illustrate that an interface can be used instead of an abstract class to process Shapes polymorphically. Notice that the output for the program (Fig. 9.31) is identical to Fig. 9.22. Also, notice that Object method toString is called through a Shape interface reference (line 44).

Software Engineering Observation 9.31

All methods of class Object can be called by using a reference of an interface data type—a reference refers to an object, and all objects have the methods defined by class Object.

One benefit of using interfaces is that a class can implement as many interfaces as it needs in addition to extending a class. To implement more than one interface, simply provide a comma-separated list of interface names after keyword implements in the class definition. This is particularly useful in the GUI event-handling mechanism. A class that implements more than one event-listener interface (such as ActionListener in earlier

```java
32 // get y coordinate
33 public int getY()
34 {
35 return y;
36 }
37
38 // convert point into String representation
39 public String toString()
40 {
41 return "[" + x + ", " + y + "]";
42 }
43
44 // calculate area
45 public double area()
46 {
47 return 0.0;
48 }
49
50 // calculate volume
51 public double volume()
52 {
53 return 0.0;
54 }
55
56 // return shape name
57 public String getName()
58 {
59 return "Point";
60 }
61
62 } // end class Point
```

Fig. 9.28 Point implementation of interface Shape (part 2 of 2).
examples) can process different types of GUI events, as we will see in Chapter 12 and Chapter 13.

```java
// Fig. 9.29: Circle.java
// Definition of class Circle

public class Circle extends Point {  // inherits from Point

    protected double radius;

    // no-argument constructor
    public Circle() {
        // implicit call to superclass constructor here
        setRadius( 0 );
    }

    // constructor
    public Circle( double circleRadius, int xCoordinate,
    int yCoordinate ) {
        // call superclass constructor
        super( xCoordinate, yCoordinate );
        setRadius( circleRadius );
    }

    // set radius of Circle
    public void setRadius( double circleRadius ) {
        radius = ( circleRadius >= 0 ? circleRadius : 0 );
    }

    // get radius of Circle
    public double getRadius() {
        return radius;
    }

    // calculate area of Circle
    public double area() {
        return Math.PI * radius * radius;
    }

    // convert Circle to a string representation
    public String toString() {
        return "Center = " + super.toString() + "; Radius = " + radius;
    }
}
```

**Fig. 9.29** Circle subclass of Point—indirect implementation of interface Shape (part 1 of 2).
// return shape name
public String getName()
{
    return "Circle";
}

// end class Circle

Fig. 9.29  Circle subclass of Point—indirect implementation of interface Shape (part 2 of 2).

public class Cylinder extends Circle {
    protected double height; // height of Cylinder

    // no-argument constructor
    public Cylinder()
    {
        // implicit call to superclass constructor here
        setHeight( 0 );
    }

    // constructor
    public Cylinder( double cylinderHeight, double cylinderRadius, int xCoordinate, int yCoordinate )
    {
        // call superclass constructor
        super( cylinderRadius, xCoordinate, yCoordinate );

        setHeight( cylinderHeight );
    }

    // set height of Cylinder
    public void setHeight( double cylinderHeight )
    {
        height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
    }

    // get height of Cylinder
    public double getHeight()
    {
        return height;
    }

    // calculate area of Cylinder (i.e., surface area)
    public double area()
    {
        // Fig. 9.30: Cylinder.java
        // Definition of class Cylinder.

        public class Cylinder extends Circle {
            protected double height; // height of Cylinder

            // no-argument constructor
            public Cylinder()
            {
                // implicit call to superclass constructor here
                setHeight( 0 );
            }

            // constructor
            public Cylinder( double cylinderHeight, double cylinderRadius, int xCoordinate, int yCoordinate )
            {
                // call superclass constructor
                super( cylinderRadius, xCoordinate, yCoordinate );

                setHeight( cylinderHeight );
            }

            // set height of Cylinder
            public void setHeight( double cylinderHeight )
            {
                height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
            }

            // get height of Cylinder
            public double getHeight()
            {
                return height;
            }

            // calculate area of Cylinder (i.e., surface area)
            public double area()
            {

Fig. 9.30  Cylinder subclass of Circle—indirect implementation of interface Shape (part 1 of 2).
return 2 * super.area() + 2 * Math.PI * radius * height;
}

// calculate volume of Cylinder
public double volume()
{
    return super.area() * height;
}

// convert Cylinder to a String representation
public String toString()
{
    return super.toString() + "; Height = " + height;
}

// return shape name
public String getName()
{
    return "Cylinder";
}
}  // end class Cylinder

Fig. 9.30 Cylinder subclass of Circle—indirect implementation of interface Shape (part 2 of 2).

// Fig. 9.31: Test.java
// Test Point, Circle, Cylinder hierarchy with interface Shape.

// Java core packages
import java.text.DecimalFormat;

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

public class Test {

    // test Shape hierarchy
    public static void main( String args[] )
    {
        // create shapes
        Point point = new Point( 7, 11 );
        Circle circle = new Circle( 3.5, 22, 8 );
        Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );

        // create Shape array
        Shape arrayOfShapes[] = new Shape[ 3 ];

        // aim arrayOfShapes[ 0 ] at subclass Point object
        arrayOfShapes[ 0 ] = point;

    }
}

Fig. 9.31 Shape, Point, Circle, Cylinder hierarchy (part 1 of 2).
// aim arrayOfShapes[1] at subclass Circle object
arrayOfShapes[1] = circle;

// aim arrayOfShapes[2] at subclass Cylinder object

// get name and String representation of each shape
String output =
    point.getName() + ": " + point.toString() + 
    circle.getName() + ": " + circle.toString() + 
    cylinder.getName() + ": " + cylinder.toString();

DecimalFormat precision2 = new DecimalFormat("0.00");

// loop through arrayOfShapes and get name,
// area and volume of each shape in arrayOfShapes
for ( int i = 0; i < arrayOfShapes.length; i++ ) {
    output += 
    arrayOfShapes[i].getName() + 
    ": " + arrayOfShapes[i].toString() + 
    "Area = " + precision2.format(arrayOfShapes[i].area()) + 
    "Volume = " + precision2.format(arrayOfShapes[i].volume());
}

JOptionPane.showMessageDialog(null, output,
    "Demonstrating Polymorphism",
    JOptionPane.INFORMATION_MESSAGE);
System.exit(0);
} // end class Test

Fig. 9.31 Shape, Point, Circle, Cylinder hierarchy (part 2 of 2).
Another use of interfaces is to define a set of constants that can be used in many class definitions. Consider interface `Constants`

```java
public interface Constants {
    public static final int ONE = 1;
    public static final int TWO = 2;
    public static final int THREE = 3;
}
```

Classes that implement interface `Constants` can use `ONE`, `TWO` and `THREE` anywhere in the class definition. A class can even use these constants by importing the interface, then referring to each constant as `Constants.ONE`, `Constants.TWO` and `Constants.THREE`. There are no methods declared in this interface, so a class that implements the interface is not required to provide any method implementations.

### 9.20 Inner Class Definitions

All the class definitions discussed so far were defined at file scope. For example, if a file contained two classes, one class was not nested in the body of the other class. Java provides a facility called `inner classes`, in which classes can be defined inside other classes. Such classes can be complete class definitions or `anonymous inner class` definitions (classes without a name). Inner classes are used mainly in event handling. However, they have other benefits. For example, the implementation of the queue abstract data type discussed in Section 8.16.1 might use an inner class to represent the objects that store each item currently in the queue. Only the queue data structure requires knowledge of how the objects are stored internally, so the implementation can be hidden by defining an inner class as part of class `Queue`.

Inner classes frequently are used with GUI event handling, we take this opportunity not only to show you inner class definitions, but also to demonstrate an application that executes in its own window. After you complete this example, you will be able to use in your applications the GUI techniques shown only in applets so far.

To demonstrate an inner class definition, Fig. 9.33 uses a simplified version of the `Time3` class (renamed `Time` in Fig. 9.32) from Figure 8.8. Class `Time` provides a default constructor, the same `set/get` methods as Figure 8.8 and a `toString` method. Also, this program defines class `TimeTestWindow` as an application. The application executes in its own window. [Note: We do not discuss class `Time` here, because all its features were discussed in Chapter 8.]

```java
// Fig. 9.32: Time.java
// Time class definition.

// Java core packages
import java.text.DecimalFormat;

// This class maintains the time in 24-hour format
public class Time extends Object {
    private int hour;     // 0 - 23
    private int minute;   // 0 - 59
    private int second;   // 0 - 59

    // Fig. 9.32 Time class (part 1 of 3).
```
public Time()
{
    setTime( 0, 0, 0 );
}

public void setTime( int hour, int minute, int second )
{
    setHour( hour );
    setMinute( minute );
    setSecond( second );
}

public void setHour( int h )
{
    hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
}

public void setMinute( int m )
{
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
}

public void setSecond( int s )
{
    second = ( ( s >= 0 && s < 60 ) ? s : 0 );
}

public int getHour()
{
    return hour;
}

public int getMinute()
{
    return minute;
}

public int getSecond()
{
    return second;
}
// convert to String in standard-time format
public String toString()
{
  DecimalFormat twoDigits = new DecimalFormat( "00" );
  return ( (getHour() == 12 || getHour() == 0 ) ?
    12 : getHour() % 12 ) + ":" +
    twoDigits.format( getMinute() ) + ":" +
    twoDigits.format( getSecond() ) +
    ( getHour() < 12 ? " AM" : " PM" );
}
} // end class Time

Fig. 9.32  Time class (part 3 of 3).

// Fig. 9.33: TimeTestWindow.java
// Demonstrating the Time class set and get methods

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

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

public class TimeTestWindow extends JFrame {
  private Time time;
  private JLabel hourLabel, minuteLabel, secondLabel;
  private JTextField hourField, minuteField, secondField, displayField;
  private JButton exitButton;

  // set up GUI
  public TimeTestWindow()
  {
    super( "Inner Class Demonstration" );
    time = new Time();

    // create an instance of inner class ActionEventHandler
    ActionEventHandler handler = new ActionEventHandler();

    // set up GUI
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );
    hourLabel = new JLabel( "Set Hour" );
    hourField = new JTextField( 10 );
    hourField.addActionListener( handler );
    hourField.setEditable( false );
    hourField.addActionListener( handler );
    hourField.addFocusListener( handler );
    hourField.setText( String.valueOf( time.getHour() ) );
    hourLabel.add( hourField );

    minuteLabel = new JLabel( "Set Hour" );
    minuteField = new JTextField();
    minuteField.setEditable( false );
    minuteField.addActionListener( handler );
    minuteField.addFocusListener( handler );
    minuteField.setText( String.valueOf( time.getMinute() ) );
    minuteLabel.add( minuteField );

    secondLabel = new JLabel( "Set Hour" );
    secondField = new JTextField();
    secondField.setEditable( false );
    secondField.addActionListener( handler );
    secondField.addFocusListener( handler );
    secondField.setText( String.valueOf( time.getSecond() ) );
    secondLabel.add( secondField );

    displayField = new JTextField();
    displayField.setEditable( false );
    displayField.addActionListener( handler );
    displayField.addFocusListener( handler );
    displayField.setText( time.toString() );
    displayField.setSize( 40, 20 );
    displayField.add( new JScrollPane( displayField ) );

    // add components to JFrame
    add( hourLabel );
    add( hourField );
    add( minuteLabel );
    add( minuteField );
    add( secondLabel );
    add( secondField );
    add( displayField );

    // set up action event handler
    handler.addActionListener( this );

    // show the JFrame
    setVisible( true );
    setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    setSize( 310, 300 );
    setLocationRelativeTo( null );
  }

  public static void main( String[] args )
  {
    new TimeTestWindow();
  }

  public void actionPerformed( ActionEvent event )
  {
    // new time from user input
    int hour = Integer.parseInt( hourField.getText() );
    int minute = Integer.parseInt( minuteField.getText() );
    int second = Integer.parseInt( secondField.getText() );

    // set the new time
    time.set( hour, minute, second );

    // update display
    displayField.setText( time.toString() );
  }
}

Fig. 9.33  Demonstrating an inner class in a windowed application (part 1 of 4).
container.add( hourLabel );
container.add( hourField );

minuteLabel = new JLabel( "Set minute" );
minuteField = new JTextField( 10 );
minuteField.addActionListener( handler );
container.add( minuteLabel );
container.add( minuteField );

secondLabel = new JLabel( "Set Second" );
secondField = new JTextField( 10 );
secondField.addActionListener( handler );
container.add( secondLabel );
container.add( secondField );

displayField = new JTextField( 30 );
displayField.setEditable( false );
container.add( displayField );

exitButton = new JButton( "Exit" );
exitButton.addActionListener( handler );
container.add( exitButton );

} // end constructor

// display time in displayField
public void displayTime()
{
displayField.setText( "The time is: " + time );
}

// create TimeTestWindow and display it
public static void main( String args[] )
{
TimeTestWindow window = new TimeTestWindow();

window.setSize( 400, 140 );
window.setVisible( true );
}

// inner class definition for handling JTextField and
// JButton events
private class ActionEventHandler
implements ActionListener {

// method to handle action events
public void actionPerformed( ActionEvent event )
{
    // user pressed exitButton
    if ( event.getSource() == exitButton )
        System.exit( 0 );   // terminate the application
}
```java
87 // user pressed Enter key in hourField
88 else if ( event.getSource() == hourField ) {
89  time.setHour(
90     Integer.parseInt( event.getActionCommand() ) );
91  hourField.setText( "" );
92 }
93
94 // user pressed Enter key in minuteField
95 else if ( event.getSource() == minuteField ) {
96  time.setMinute(
97     Integer.parseInt( event.getActionCommand() ) );
98  minuteField.setText( "" );
99 }
100
101 // user pressed Enter key in secondField
102 else if ( event.getSource() == secondField ) {
103  time.setSecond(
104     Integer.parseInt( event.getActionCommand() ) );
105  secondField.setText( "" );
106 }
107
108  displayTime();
109 }
110
111 } // end inner class ActionEventHandler
112
113 } // end class TimeTestWindow
```

**Fig. 9.33** Demonstrating an inner class in a windowed application (part 3 of 4).
In Fig. 9.33, line 11 indicates that class **TimeTestWindow** extends class **JFrame** (from package *javax.swing*) rather than class **JApplet** (as shown in Figure 8.8). Superclass **JFrame** provides the basic attributes and behaviors of a window—a *title bar* and buttons to *minimize*, *maximize* and *close* the window (all labeled in the first screen capture). Class **TimeTestWindow** uses the same GUI components as the applet of Fig. 8.8, except that the button (line 16) is now called **exitButton** and is used to terminate the application.

The *init* method of the applet has been replaced by a constructor (lines 19–54) to guarantee that the window’s GUI components are created as the application begins executing. Method **main** (lines 67–73) defines a *new* object of class **TimeTestWindow** that results in a call to the constructor. Remember, *init* is a special method that is guaranteed to be called when an applet begins execution. However, this program is not an applet, so if we did define the *init* method, it would not be called automatically.

Several new features appear in the constructor. Line 21 calls the superclass **JFrame** constructor with the string "*Inner Class Demonstration*". This string is displayed in the title bar of the window by class **JFrame**’s constructor. Line 26 defines one instance of our inner class **ActionEventHandler** (defined at lines 77–111) and assigns it to **handler**. This reference is passed to each of the four calls to **addActionListener** (lines 34, 40, 46 and 55) that register the event handlers for each GUI component that generates events in this example (**hourField**, **minuteField**, **secondField** and **exitButton**). Each call to **addActionListener** requires an object of type...
**ActionListener** to be passed as an argument. Actually, **handler is an ActionListener**. Line 77 (the first line of the inner class definition) indicates that inner class **ActionEventHandler** implements **ActionListener**. Thus, every object of type **ActionEventHandler** is an **ActionListener**. The requirement that **addListener** be passed an object of type **ActionListener** is satisfied! The *is a* relationship is used extensively in the GUI event-handling mechanism, as you will see over the next several chapters. The inner class is defined as **private** because it will be used only in this class definition. Inner classes can be **private, protected** or **public**.

An inner class object has a special relationship with the outer class object that creates it. The inner class object is allowed to access directly all the instance variables and methods of the outer class object. The **actionPerformed** method (line 77–105) of class **ActionEventHandler** does just that. Throughout the method, the instance variables **time**, **exitButton**, **hourField**, **minuteField** and **secondField** are used, as is method **displayTime**. Notice that none of these needs a “handle” to the outer class object. This is a free relationship created by the compiler between the outer class and its inner classes.

**Software Engineering Observation 9.32**

An inner class object is allowed to access directly all the variables and methods of the outer class object that defined it.

This application must be terminated by pressing the **Exit** button. Remember, an application that displays a window must be terminated with a call to **System.exit(0)**. Also note that a window in Java is 0 pixels wide, 0 pixels tall and not displayed by default. Lines 71–72 use methods **resize** and **setVisible** to size the window and display it on the screen. These methods are defined in class **java.awt.Component** originally and inherited into class **JFrame**.

An inner class can also be defined inside a method of a class. Such an inner class has access to its outer class’s members. However, it has limited access to the local variables for the method in which it is defined.

**Software Engineering Observation 9.33**

An inner class defined in a method is allowed to access directly all the instance variables and methods of the outer class object that defined it and any **final** local variables in the method.

The application of Fig. 9.34 modifies class **TimeTestWindow** of Fig. 9.33 to use **anonymous inner classes** defined in methods. An anonymous inner class has no name, so one object of the anonymous inner class must be created at the point where the class is defined in the program. We demonstrate anonymous inner classes two ways in this example. First, we use separate anonymous inner classes that implement an interface (**ActionListener**) to create event handlers for each of the three **JTextField** **hourField**, **minuteField** and **secondField**. We also demonstrate how to terminate an application when the user clicks the Close box on the window. The event handler is defined as an anonymous inner class that extends a class (**WindowAdapter**). The **Time** class used is identical to Fig. 9.32, so it is not included here. Also, the **Exit** button has been removed from this example.
// Fig. 9.34: TimeTestWindow.java
// Demonstrating the Time class set and get methods

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

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

public class TimeTestWindow extends JFrame {
  private Time time;
  private JLabel hourLabel, minuteLabel, secondLabel;
  private JTextField hourField, minuteField,
                  secondField, displayField;

  // set up GUI
  public TimeTestWindow()
  {
    super( "Inner Class Demonstration" );

    // create Time object
    time = new Time();

    // create GUI
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );
    hourLabel = new JLabel( "Set Hour" );
    hourField = new JTextField( 10 );

    // register hourField event handler
    hourField.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent event ) {
        time.setHour( Integer.parseInt( event.getActionCommand() ) );
        hourField.setText( "" );
        displayTime();
      }
    } );

    container.add( hourLabel );
    container.add( hourField );
  }  // end call to addActionListener

  // anonymous inner class
  new ActionListener() {

    public void actionPerformed( ActionEvent event )
    {
      time.setHour( Integer.parseInt( event.getActionCommand() ) );
      hourField.setText( "" );
      displayTime();
    }
  }  // end anonymous inner class

  container.add( hourLabel );
  container.add( hourField );
}

Fig. 9.34  Demonstrating anonymous inner classes (part 1 of 4).
minuteLabel = new JLabel( "Set minute" );
minuteField = new JTextField( 10 );

// register minuteField event handler
minuteField.addActionListener(

    // anonymous inner class
    new ActionListener() {

        public void actionPerformed( ActionEvent event )
        {
            time.setMinute(
                Integer.parseInt( event.getActionCommand() ) );
            minuteField.setText( "" );
            displayTime();
        }
    }  // end anonymous inner class
); // end call to addActionListener

container.add( minuteLabel );
container.add( minuteField );

secondLabel = new JLabel( "Set Second" );
secondField = new JTextField( 10 );

secondField.addActionListener(

    // anonymous inner class
    new ActionListener() {

        public void actionPerformed( ActionEvent event )
        {
            time.setSecond(
                Integer.parseInt( event.getActionCommand() ) );
            secondField.setText( "" );
            displayTime();
        }
    }  // end anonymous inner class
); // end call to addActionListener

container.add( secondLabel );
container.add( secondField );

displayField = new JTextField( 30 );
displayField.setEditable( false );
container.add( displayField );

Fig. 9.34 Demonstrating anonymous inner classes (part 2 of 4).
105  // display time in displayField
106  public void displayTime()
107  {
108       displayField.setText( "The time is: " + time );
109  }
110
111  // create TimeTestWindow, register for its window events
112  // and display it to begin application's execution
113  public static void main( String args[] )
114  {
115       TimeTestWindow window = new TimeTestWindow();
116
117       // register listener for windowClosing event
118       window.addWindowListener(
119                // anonymous inner class for windowClosing event
120                new WindowAdapter() {
121
122                    // terminate application when user closes window
123                    public void windowClosing( WindowEvent event )
124                    {
125                           System.exit( 0 );
126                    }
127
128                } // end anonymous inner class
129
130            ); // end call to addWindowListener
131
132            window.setSize( 400, 120 );
133            window.setVisible( true );
134
135      } // end class TimeTestWindow

Fig. 9.34  Demonstrating anonymous inner classes (part 3 of 4).
Each of the three JTextFields that generate events in this program has a similar anonymous inner class to handle its events, so we discuss only the anonymous inner class for hourField here. Lines 33–48 are a call to hourField’s addActionListener method. The argument to this method must be an object that is an ActionListener (i.e., any object of a class that implements ActionListener). Lines 36–46 use special Java syntax to define an anonymous inner class and create one object of that class that is passed as the argument to addActionListener. Line 36 uses operator new to create an object. The syntax ActionListener() begins the definition of an anonymous inner class that implements interface ActionListener. This is similar to beginning a class definition with

```java
public class MyHandler implements ActionListener {
```

The parentheses after ActionListener indicate a call to the default constructor of the anonymous inner class.

The opening left brace ( { ) at the end of line 36 and the closing right brace ( } ) at line 46 define the body of the class. Lines 38–44 define the actionPerformed method that is required in any class that implements ActionListener. Method actionPerformed is called when the user presses Enter while typing in hourField.

Software Engineering Observation 9.34

When an anonymous inner class implements an interface, the class must define every method in the interface.

Method main creates one instance of class TimeTestWindow (line 115), sizes the window (line 133) and displays the window (line 134).

Windows generate a variety of events that are discussed in Chapter 13. For this example we discuss the one event generated when the user clicks the window’s close box—a window closing event. Lines 118–131 enable the user to terminate the application by
clicking the window’s close box (labeled in the first screen capture). Method `addWindowListener` registers a window event listener. The argument to `addWindowListener` must be a reference to an object that is a `WindowListener` (package `java.awt.event`) (i.e., any object of a class that implements `WindowListener`). However, there are seven different methods that must be defined in every class that implements `WindowListener` and we only need one in this example—`windowClosing`. For event handling interfaces with more than one method, Java provides a corresponding class (called an adapter class) that already implements all the methods in the interface for you. All you need to do is extend the adapter class and override the methods you require in your program.

**Common Programming Error 9.10**

Extending an adapter class and misspelling the name of the method you are overriding is a logic error.

Lines 121–129 use special Java syntax to define an anonymous inner class and create one object of that class that is passed as the argument to `addWindowListener`. Line 118 uses operator `new` to create an object. The syntax `WindowAdapter()` begins the definition of an anonymous inner class that extends class `WindowAdapter`. This is similar to beginning a class definition with

```
public class MyHandler extends WindowAdapter {
```

The parentheses after `WindowAdapter` indicate a call to the default constructor of the anonymous inner class. Class `WindowAdapter` implements interface `WindowListener`, so every `WindowAdapter` object is a `WindowListener`—the exact type required for the argument to `addWindowListener`.

The opening left brace (`{`) at the end of line 121 and the closing right brace (`}`) at line 129 define the body of the class. Lines 124–127 override the `windowClosing` method of `WindowAdapter` that is called when the user clicks the window’s close box. In this example, `windowClosing` terminates the application.

In the last two examples, we have seen that inner classes can be used to create event handlers and that separate anonymous inner classes can be defined to handle events individually for each GUI component. In Chapter 12 and Chapter 13, we revisit this concept as we discuss the event handling mechanism in detail.

### 9.21 Notes on Inner Class Definitions

This section presents several notes of interest to programmers regarding the definition and use of inner classes.

1. Compiling a class that contains inner classes results in a separate `.class` file for every class. Inner classes with names have the file name `OuterClassName$InnerClassName.class`. Anonymous inner classes have the file name `OuterClassName#.#.class`, where `#` starts at 1 and is incremented for each anonymous inner class encountered during compilation.

2. Inner classes with class names can be defined as `public`, `protected`, package access or `private` and are subject to the same usage restrictions as other members of a class.
3. To access the outer class’s this reference, use OuterClassName.this.

4. The outer class is responsible for creating objects of its inner classes. To create an object of another class’s inner class, first create an object of the outer class and assign it to a reference (we will call it ref). Then use a statement of the following form to create an inner class object:

   OuterClassName.InnerClassName innerRef = ref.new InnerClassName();

5. An inner class can be declared static. A static inner class does not require an object of its outer class to be defined (whereas a nonstatic inner class does). A static inner class does not have access to the outer class’s nonstatic members.

### 9.22 Type-Wrapper Classes for Primitive Types

Each of the primitive types has a type-wrapper class. These classes are called Character, Byte, Short, Integer, Long, Float, Double and Boolean. Each type-wrapper class enables you to manipulate primitive types as objects of class Object. Therefore, values of the primitive data types can be processed polymorphically if they are maintained as objects of the type-wrapper classes. Many of the classes we will develop or reuse manipulate and share Object. These classes cannot polymorphically manipulate variables of primitive types, but they can polymorphically manipulate objects of the type-wrapper classes, because every class ultimately is derived from class Object.

Each of the numeric classes—Byte, Short, Integer, Long, Float and Double—inherits from class Number. Each of the type wrappers is declared final, so their methods are implicitly final and may not be overridden. Note that many of the methods that process the primitive data types are defined as static methods of the type-wrapper classes. If you need to manipulate a primitive value in your program, first refer to the documentation for the type-wrapper classes—the method you need might already be defined. We will use the type-wrapper classes polymorphically in our study of data structures in Chapters 22 and 23.

### 9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation

We now examine our elevator simulator design to see whether it might benefit from inheritance. In previous chapters, we have been treating ElevatorButton and FloorButton as separate classes. These classes, however, have much in common—both have attribute pressed and behaviors pressButton and resetButton. To apply inheritance, we first look for commonality between these classes. We extract this commonality, place it into superclass Button, then derive subclasses ElevatorButton and FloorButton from Button.

Let us now examine the similarities between classes ElevatorButton and FloorButton. Figure 9.35 shows the attributes and operations of each class. Both classes have their attribute (pressed) and operations (pressButton and resetButton) in common.
We must now examine whether these objects exhibit distinct behavior. If the ElevatorButton and FloorButton objects have the same behavior, then we cannot justify using two separate classes to instantiate these objects. However, if these objects have distinct behaviors, we can place the common attributes and operations in a superclass Button; then ElevatorButton and FloorButton inherit both the attributes and operations of Button.

We have been treating the FloorButton as if it behaves differently from the ElevatorButton—the FloorButton requests the Elevator to move to the Floor of the request, and the ElevatorButton signals the Elevator to move to the opposite Floor. Under closer scrutiny, we notice that both the FloorButton and the ElevatorButton signal the Elevator to move to a Floor, and the Elevator decides whether to move. The Elevator will sometimes move in response to a signal from the FloorButton, and the Elevator will always move in response to a signal from the ElevatorButton—however, neither the FloorButton nor the ElevatorButton decides for the Elevator that the Elevator must move to the other Floor. We conclude that both FloorButton and ElevatorButton have the same behavior—both signal the Elevator to move—so we combine (not inherit) classes into one Button class and discard class FloorButton and ElevatorButton from our case study.

We turn our attention to classes ElevatorDoor and FloorDoor. Figure 9.36 shows the attributes and operations of classes ElevatorDoor and FloorDoor—these two classes are structurally similar to each other as well, because both classes possess the attribute open and the two operations openDoor and closeDoor. In addition, both the ElevatorDoor and FloorDoor have the same behavior—they inform a Person that a door in the simulation has opened. The Person then decides either to enter or exit the Elevator, depending on which door opened. In other words, neither the ElevatorDoor nor the FloorDoor decides for the Person that the Person must enter or exit the Elevator. We combine both ElevatorDoor and FloorDoor into one Door class and eliminate classes ElevatorDoor and FloorDoor from our case study.²

![Fig. 9.35 Attributes and operations of classes FloorButton and ElevatorButton.]

We must now examine whether these objects exhibit distinct behavior. If the ElevatorButton and FloorButton objects have the same behavior, then we cannot justify using two separate classes to instantiate these objects. However, if these objects have distinct behaviors, we can place the common attributes and operations in a superclass Button; then ElevatorButton and FloorButton inherit both the attributes and operations of Button.

We have been treating the FloorButton as if it behaves differently from the ElevatorButton—the FloorButton requests the Elevator to move to the Floor of the request, and the ElevatorButton signals the Elevator to move to the opposite Floor. Under closer scrutiny, we notice that both the FloorButton and the ElevatorButton signal the Elevator to move to a Floor, and the Elevator decides whether to move. The Elevator will sometimes move in response to a signal from the FloorButton, and the Elevator will always move in response to a signal from the ElevatorButton—however, neither the FloorButton nor the ElevatorButton decides for the Elevator that the Elevator must move to the other Floor. We conclude that both FloorButton and ElevatorButton have the same behavior—both signal the Elevator to move—so we combine (not inherit) classes into one Button class and discard class FloorButton and ElevatorButton from our case study.

We turn our attention to classes ElevatorDoor and FloorDoor. Figure 9.36 shows the attributes and operations of classes ElevatorDoor and FloorDoor—these two classes are structurally similar to each other as well, because both classes possess the attribute open and the two operations openDoor and closeDoor. In addition, both the ElevatorDoor and FloorDoor have the same behavior—they inform a Person that a door in the simulation has opened. The Person then decides either to enter or exit the Elevator, depending on which door opened. In other words, neither the ElevatorDoor nor the FloorDoor decides for the Person that the Person must enter or exit the Elevator. We combine both ElevatorDoor and FloorDoor into one Door class and eliminate classes ElevatorDoor and FloorDoor from our case study.²

². As we continue to discuss inheritance throughout this section, we refer to the class diagram of Fig. 3.23 to determine similarities among objects. However, this diagram contains classes FloorButton and ElevatorButton, which we have eliminated recently from the case study. Therefore, during our discussion of inheritance, when we mention the FloorButton, we refer to that Floor’s Button object, and when we mention the ElevatorButton, we refer to the Elevator’s Button object.
In Section 4.14, we encountered the problem of representing the location of the Person—on what Floor is the Person located when riding in the Elevator? Using inheritance, we may now model a solution. Both the Elevator and the two Floors are locations at which the Person exists in the simulator. In other words, the Elevator and the Floors are types of locations. Therefore, classes Elevator and Floor should inherit from an abstract superclass called Location. We declare class Location as abstract, because, for the purposes of our simulation, a location is too generic a term to define a real object. The UML requires that we place abstract class names (and abstract methods) in italics. Superclass Location contains the protected attribute locationName, which contains a String value of "firstFloor", "secondFloor" or "elevator". Therefore, only classes Elevator and Floor have access to locationName. In addition, we include the public method getLocationName to return the name of the location.

We search for more similarities between classes Floor and Elevator. First of all, according to the class diagram of Fig. 3.23, Elevator contains a reference to both a Button and a Door—the ElevatorButton (the Elevator’s Button) and the ElevatorDoor (the Elevator’s Door). Class Floor, through its association with class ElevatorShaft (class ElevatorShaft “connects” class Floor), also contains a reference to a Button and a Door—the FloorButton (that Floor’s Button) and the FloorDoor (that Floor’s Door). Therefore, in our simulation, the Location class, which is the superclass of classes Elevator and Floor, will contain public methods getButton and getDoor, which return a Button or Door reference, respectively. Class Floor overrides these methods to return the Button and Door references of that Floor, and class Elevator overrides these methods to return the Button and Door references of the Elevator. In other words, class Floor and Elevator exhibit distinct behavior from each other but share attribute locationName and methods getButton and getDoor—therefore, we may use inheritance for these classes.

The UML offers a relationship called a generalization to model inheritance. Figure 9.35 is the generalization diagram of superclass Location and subclasses Elevator and Floor. The empty-head arrows specify that classes Elevator and Floor inherit from class Location. Note that attribute locationName has an access modifier that we have not yet seen—the pound sign (#), indicating that locationName is a protected member, so Location subclasses may access this attribute. Note that classes Floor and Elevator contain additional attributes and methods that further distinguish these classes.
Classes `Floor` and `Elevator` override methods `getButton` and `getDoor` from their superclass `Location`. In Fig. 9.37, these methods are italicized in class `Location`, indicating that they are *abstract methods*. However, the methods in the subclasses are not italicized—these methods became *concrete methods* by overriding the abstract methods.

Class `Person` now contains a `Location` object representing whether the `Person` is on the first or second `Floor` or inside the `Elevator`. We remove the association between `Person` and `Elevator` and the association between `Person` and `Floor` from the class diagram, because the `Location` object acts as both the `Elevator` and `Floor` reference for the `Person`. A `Person` sets its `Location` object to reference the `Elevator` when that `Person` enters the `Elevator`. A `Person` sets its `Location` object to reference a `Floor` when the `Person` is on that `Floor`. Lastly, we assign class `Elevator` two `Location` objects to represent the `Elevator`’s reference to the current `Floor` and destination `Floor` (we originally used integers to describe these references). Figure 9.38 provides an updated class diagram of our model by incorporating inheritance and eliminating classes `FloorButton`, `ElevatorButton`, `FloorDoor` and `ElevatorDoor`.

---

**Fig. 9.37** Generalization diagram of superclass `Location` and subclasses `Elevator` and `Floor`.

<table>
<thead>
<tr>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td># locationName : String</td>
</tr>
<tr>
<td>+ getLocationName() : String</td>
</tr>
<tr>
<td>+ getButton() : Button</td>
</tr>
<tr>
<td>+ getDoor() : Door</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Elevator</th>
</tr>
</thead>
<tbody>
<tr>
<td>- moving : Boolean = false</td>
</tr>
<tr>
<td>- summoned: Boolean = false</td>
</tr>
<tr>
<td>- currentFloor : Integer</td>
</tr>
<tr>
<td>- destinationFloor : Integer</td>
</tr>
<tr>
<td>- capacity : Integer = 1</td>
</tr>
<tr>
<td>- travelTime : Integer = 5</td>
</tr>
<tr>
<td># locationName : String</td>
</tr>
<tr>
<td>+ ride() : void</td>
</tr>
<tr>
<td>+ requestElevator() : void</td>
</tr>
<tr>
<td>+ enterElevator() : void</td>
</tr>
<tr>
<td>+ exitElevator() : void</td>
</tr>
<tr>
<td>+ departElevator() : void</td>
</tr>
<tr>
<td>+ getLocationName() : String</td>
</tr>
<tr>
<td>+ getButton() : Button</td>
</tr>
<tr>
<td>+ getDoor() : Door</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Floor</th>
</tr>
</thead>
<tbody>
<tr>
<td>- capacity : Integer = 1</td>
</tr>
<tr>
<td># locationName : String</td>
</tr>
<tr>
<td>+ getLocationName() : String</td>
</tr>
<tr>
<td>+ getButton() : Button</td>
</tr>
<tr>
<td>+ getDoor() : Door</td>
</tr>
</tbody>
</table>
We have allowed a \texttt{Person}, occupying a \texttt{Location}, to interact with several of the objects in the simulator. For example, a \texttt{Person} can press a \texttt{Button} or be informed when a \texttt{Door} opens from that \texttt{Person}'s specific \texttt{Location}. In addition, because a \texttt{Person} may occupy only one \texttt{Location} at a time, that \texttt{Person} may interact with only the objects known to that \texttt{Location}—a \texttt{Person} will never perform an illegal action, such as pressing the first \texttt{Floor}'s \texttt{Button} while riding the \texttt{Elevator}.

We presented the class attributes and operations with access modifiers in Fig. 8.13. Now, we present a modified diagram incorporating inheritance in Fig. 9.39.
Class `Elevator` now contains two `Location` objects, called `currentFloor` and `destinationFloor`, replacing the integer values we used previously to describe the `Floors`. Lastly, `Person` contains a `Location` object named `location`, which indicates whether `Person` is on a `Floor` or in the `Elevator`.

**Implementation: Forward Engineering (Incorporating Inheritance)**

“Thinking About Objects” Section 8.17 used the UML to express the Java class structure for our simulation. We continue our implementation while incorporating inheritance, using class `Elevator` as an example. For each class in the class diagram of Fig. 9.38,
1. If a class $A$ is a subclass of class $B$, then $A$ extends $B$ in the class declaration and calls the constructor of $B$. For example, class $\text{Elevator}$ is a subclass of abstract superclass $\text{Location}$, so the class declaration should read

```java
public class Elevator extends Location {

    public Elevator
    {
        super();
    }

    ...
}
```

2. If class $B$ is an abstract class and class $A$ is a subclass of class $B$, then class $A$ must override the abstract methods of class $B$ (if class $A$ is to be a concrete class). For example, class $\text{Location}$ contains abstract methods $\text{getLocationName}$, $\text{getButton}$ and $\text{getDoor}$, so class $\text{Elevator}$ must override these methods (note that $\text{getButton}$ returns the $\text{Elevator}$’s $\text{Button}$ object, and $\text{getDoor}$ returns the $\text{Elevator}$’s $\text{Door}$ object—$\text{Elevator}$ contains associations with both objects, according to the class diagram of Fig. 9.38).

```java
public class Elevator extends Location {

    // class attributes
    private boolean moving;
    private boolean summoned;
    private Location currentFloor;
    private Location destinationFloor;
    private int capacity = 1;
    private int travelTime = 5;

    // class objects
    private Button elevatorButton;
    private Door elevatorDoor;
    private Bell bell;

    // class constructor
    public Elevator()
    {
        super();
    }

    // class methods
    public void ride() {}
    public void requestElevator() {}
    public void enterElevator() {}
    public void exitElevator() {}
    public void departElevator() {

        // method overriding $\text{getLocationName}$
        public String getLocationName()
        {
            return "elevator";
        }
    }
}
```
Using forward engineering provides a sound beginning for code implementation in any language. “Thinking About Objects” Section 11.10 returns to interactions and focuses on how objects generate and handle the messages passed in collaborations. In addition, we forward engineer more class diagrams to implement these interactions. Eventually, we present the Java code for our simulator in Appendix G, Appendix H and Appendix I.

9.24 (Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

Now that we have introduced object-oriented programming, we begin our deeper presentation of design patterns. In Section 1.17, we mentioned that the “gang of four” described 23 design patterns using three categories—creational, structural and behavioral. In this and the remaining “Discovering Design Patterns” sections, we discuss the design patterns of each type and their importance, and we mention how each pattern relates to the Java material in the book. For example, several Java Swing components that we introduce in Chapters 12 and 13 use the Composite design pattern, so we introduce the Composite design pattern in Section 13.18. Figure 9.40 identifies the 18 gang-of-four design patterns discussed in this book.

Figure 9.40 lists 18 of the most widely used patterns in the software-engineering industry. There are many popular patterns that have been documented since the gang-of-four book—these include the concurrent design patterns, which are especially helpful in the design of multithreaded systems. Section 15.13 discusses some of these patterns used in industry. Architectural patterns, as we discuss in Section 17.10, specify how subsystems interact with each other. Figure 9.41 specifies the concurrency patterns and architectural patterns that we discuss in this book.
9.24.1 Creational Design Patterns

Creational design patterns address issues related to the creation of objects, such as preventing a system from creating more than one object of a class (the Singleton creational design pattern) or deferring until execution time the decision as to what types of objects are going to be created (the purpose of the other creational design patterns discussed here). 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 shapes the user will choose to draw. Based on user input, this program should be able to determine the class from which to instantiate an object. If the user creates a cylinder in the GUI, our program should “know” to instantiate an object of class Cylinder. When the user decides what geometric object to draw, the program should determine the specific subclass from which to instantiate that object.

The gang-of-four book describes five creational patterns (four of which we discuss in this book):

- Abstract Factory (Section 17.10)
- Builder (not discussed)
• Factory Method (Section 13.18)
• Prototype (Section 21.12)
• Singleton (Section 9.24)

**Singleton**

Occasionally, a system should contain exactly one object of a class—that is, once the program instantiates that object, the program should not be allowed to create additional objects of that class. For example, some systems connect to a database using only one object that manages database connections, which ensures that other objects cannot initialize unnecessary connections that would slow the system. The *Singleton design pattern* guarantees that a system instantiates a maximum of one object of a class.

Figure 9.42 demonstrates Java code using the Singleton design pattern. Line 5 declares class `Singleton` as `final`, so methods in this class cannot be overridden to handle multiple instantiations. Lines 11–14 declare a `private` constructor—only class `Singleton` can instantiate a `Singleton` object using this constructor. The `static` method `getSingletonInstance` (lines 17–24) allows the one-time instantiation of a `static` `Singleton` object (declared on line 8) by calling the `private` constructor. If the `Singleton` object has been created, line 23 merely returns a reference to the previously instantiated `Singleton` object.

Lines 10–11 of class `SingletonExample` (Fig. 9.43) declare two references to `Singleton` objects—`firstSingleton` and `secondSingleton`. Lines 14–15 call method `getSingletonInstance` and assign `Singleton` references to `firstSingleton` and `secondSingleton`, respectively. Line 18 tests if these objects reference the same `Singleton` object. Figure 9.44 shows that `firstSingleton` and `secondSingleton` share the same reference to the `Singleton` object, because each time method `getSingletonInstance` is called, it returns a reference to the same `Singleton` object.

```java
1 // Singleton.java
2 // Demonstrates Singleton design pattern
3 package com.deitel.jhtp4.designpatterns;
4
5 public final class Singleton {
6
7     // Singleton object returned by method getSingletonInstance
8     private static Singleton singleton;
9
10    // constructor prevents instantiation from other objects
11    private Singleton()
12    {
13        System.err.println("Singleton object created.");
14    }
15
16    // create Singleton and ensure only one Singleton instance
17    public static Singleton getSingletonInstance()
18    {
```

**Fig. 9.42** Class `Singleton` ensures that only one object of its class is created (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
9.24.2 Structural Design Patterns

Structural design patterns describe common ways to organize classes and objects in a system. The gang-of-four book describes seven structural design patterns (six of which we discuss in this book):

```java
// SingletonExample.java
// Attempt to create two Singleton objects
package com.deitel.jhtp4.designpatterns;

public class SingletonExample {

    // run SingletonExample
    public static void main( String args[] )
    {
        Singleton firstSingleton;
        Singleton secondSingleton;

        // create Singleton objects
        firstSingleton = SingletongetSingletonInstance();
        secondSingleton = SingletongetSingletonInstance();

        // the "two" Singletons should refer to same Singleton
        if ( firstSingleton == secondSingleton )
            System.out.println( "firstSingleton and "+
                                "secondSingleton refer to the same Singleton "+
                                "object" );
    }
}
```

**Fig. 9.43** Class `SingletonExample` attempts to create `Singleton` object more than once.

**Fig. 9.44** Class `SingletonExample` output shows that the `Singleton` object may be created only once.
Proxy
An applet should always display something while images load. Whether that “something” is a smaller image or a string of text informing the user that the images are loading, the Proxy design pattern can be applied to achieve this effect. This pattern allows one object to act as a replacement for another. Consider loading several large images (several megabytes) in a Java applet. Ideally, we would like to see these images instantaneously—however, loading large images into memory can take time to complete (especially on a slow processor). The Proxy design pattern allows the system to use one object—called a proxy object—in place of another. In our example, the proxy object could be a gauge that informs the user of what percentage of a large image has been loaded. When this image finishes loading, the proxy object is no longer needed—the applet can then display an image instead of the proxy.

9.24.3 Behavioral Design Patterns
There are many different examples of behavioral design patterns, which provide proven strategies to model how objects collaborate with one another in a system and offer special behaviors appropriate for a wide variety of applications. Let us consider the Observer behavioral design pattern—a classic example of a design pattern illustrating collaborations between objects. For example, GUI components collaborate with their listeners to respond to user interactions. GUI components use this pattern to process user interface events. A listener observes state changes in a particular GUI component by registering to handle that GUI component’s events. When the user interacts with that GUI component, the component notifies its listeners (also known as its observers) that the GUI component’s state has changed (e.g., a button has been pressed).

Another pattern we consider is the Memento behavioral design pattern—an example of offering special behavior for a wide variety of applications. The Memento pattern enables a system to save an object’s state, so that state can be restored at a later time. For example, many applications provide an “undo” capability that allows users to revert to previous versions of their work.

The gang-of-four book describes 11 behavioral design patterns (eight of which we discuss in this book):

- Chain-of-Responsibility (Section 13.18)
- Command (Section 13.18)
- Interpreter (not discussed)
- Iterator (Section 21.12)
Memento
Consider a painting program. This type of program allows a user to create graphics. Occasionally the user may position a graphic improperly in the drawing area. Painting programs offer an “undo” feature that allows the user to unwind such an error. Specifically, the program restores the drawing area’s state to that before the user placed the graphic. More sophisticated painting 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 object can be restored 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 painting-program example, the drawing area acts as the originator, because it contains attribute information describing its state—when the program first executes, the area contains no elements. The memento object stores a copy of all attributes associated with the originator’s state—i.e., the memento saves the drawing area’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 contains different information describing its state—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.

State
In certain designs, we must convey an object’s state information or represent the various states that an object can occupy. Our optional elevator simulation case study in the “Thinking About Objects” sections uses the State design pattern. Our simulation includes an elevator that moves between floors in a two-story building. A person walks across a floor and rides the elevator to the other floor. Originally, we used an integer value to represent on which floor the person is walking. However, we encountered a problem when we tried to answer the question “on what floor is the person when riding the elevator?” Actually, the person is located on neither floor—rather the person is located inside the elevator. We also realized that the elevator and the floors are locations that the person can occupy in our simulation. We created an abstract superclass called Location to represent a “location.” Subclasses Elevator and Floor inherit from superclass Location. Class Person contains a reference to a Location object, which represents the current location—eleva-
tor, first floor or second floor—of that person. Because a superclass reference can hold a subclass reference, the person’s Location attribute references the appropriate Floor object when that person is on a floor and references the Elevator object when that person is inside the elevator.

The elevator and floors contain buttons. (The elevator’s button signals the elevator to move to the other floor, and the floors’ buttons summon the elevator to the floor of the request.) Because all locations in our simulation contain buttons, class Location provides abstract method getButton. Class Elevator implements method getButton to return a reference to the Button object inside the elevator, and class Floor implements method getButton to return a reference to the Button object on the floor. Using its Location reference, the person is able to press the correct button—i.e., the person will not press a floor’s button when inside the elevator and will not press the elevator’s button when on a floor.

The State design pattern uses an abstract superclass—called the State class—which contains methods that describe behaviors for states that an object (called the context object) can occupy. In our elevator simulation, the State class is superclass Location, and the context object is the object of class Person. Note that class Location does not describe all states of class Person (e.g., whether that person is walking or waiting for the elevator)—class Location describes only the location the Person and contains method getButton so the Person can access the Button object at various locations.

A State subclass, which extends the State class, represents an individual state that the context can occupy. The State subclasses in our simulation are classes Elevator and Floor. Each State subclass contains methods that implement the State class’ abstract methods. For example, both classes Elevator and Floor implement method getButton.

The context contains exactly one reference to an object of the State class—this reference is called the state object. In the simulation, the state object is the object of class Location. When the context changes state, the state object references the State subclass object associated with that new state. For example, when the person walks from the floor into the elevator, the Person object’s Location is changed from referencing one of the Floor objects to referencing the Elevator object. When the person walks onto the floor from the elevator, the Person object’s Location references the appropriate Floor object.

### 9.24.4 Conclusion

In “Discovering Design Patterns” Section 9.24, we listed the three types of design patterns introduced in the gang-of-four book, we identified 18 of these design patterns that we discuss in this book and we discussed specific design patterns, including Singleton, Proxy, Memento and State. In “Discovering Design Patterns” Section 13.18, we introduce some design patterns associated with AWT and Swing GUI components. After reading this section, you should understand better how Java GUI components take advantage of design patterns.

### 9.24.5 Internet and World-Wide-Web Resources

The following URLs provide further information on the nature, importance and applications of design patterns.
**Design Patterns**

[www.hillside.net/patterns](http://www.hillside.net/patterns)
This page displays 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 introduces the importance of design patterns.

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

[www.enteract.com/~bradapp/javapats.html](http://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/](http://www.meurrens.org/ip-Links/java/designPatterns/)
This site displays numerous links to resources and information on Java design patterns.

**Design Patterns in C++ & Visual Basic**

[journal.iftech.com/articles/9904_shankel_patterns/](http://journal.iftech.com/articles/9904_shankel_patterns/)
This site provides insight to design patterns (the Iterator design pattern, in particular) in C++.

[mspress.microsoft.com/prod/books/sampchap/2322.htm](http://mspress.microsoft.com/prod/books/sampchap/2322.htm)

**Architectural Patterns**

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.oottips.org/mvc-pattern.html](http://www.oottips.org/mvc-pattern.html)
This site provides information and tips on using MVC.

[www.ftech.co.uk/~honeyg/articles/pda.htm](http://www.ftech.co.uk/~honeyg/articles/pda.htm)
This site contains an article on the importance of architectural patterns in software.

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

- One of the keys to the power of object-oriented programming is achieving software reusability through inheritance.

- Through inheritance, a new class inherits the instance variables and methods of a previously defined superclass. In this case, the new class is referred to as a subclass.

- With single inheritance, a class is derived from one superclass. With multiple inheritance, a subclass inherits from multiple superclasses. Java does not support multiple inheritance, but Java does provide the notion of interfaces, which offer many of the benefits of multiple inheritance.

- A subclass normally adds instance variables and methods of its own, so a subclass generally is larger than its superclass. A subclass represents a smaller set of more specific objects than its superclass.

- A subclass cannot access the private members of its superclass. A subclass can, however, access the public, protected and package access members of its superclass. The subclass must be in the superclass’s package to use superclass members with package access.

- A subclass constructor always calls the constructor for its superclass first (either explicitly or implicitly) to create and initialize the subclass members inherited from the superclass.

- Inheritance enables software reusability, which saves time in development and encourages the use of previously proven and debugged high-quality software.

- An object of a subclass can be treated as an object of its corresponding superclass, but the reverse is not true.

- A superclass exists in a hierarchical relationship with its subclasses.

- When a class is used with the mechanism of inheritance, it becomes either a superclass that supplies attributes and behaviors to other classes or a subclass that inherits those attributes and behaviors.

- An inheritance hierarchy can be arbitrarily deep within the physical limitations of a particular system, but most inheritance hierarchies have only a few levels.

- Hierarchies are useful for understanding and managing complexity. With software becoming increasingly complex, Java provides mechanisms for supporting hierarchical structures through inheritance and polymorphism.

- Modifier protected serves as an intermediate level of protection between public access and private access. Superclass protected members may be accessed by methods of the superclass, by methods of subclasses and by methods of classes in the same package.

- A superclass may be either a direct superclass or an indirect superclass. A direct superclass is the class that a subclass explicitly extends. An indirect superclass is inherited from several levels up the class hierarchy tree.

- When a superclass member is inappropriate for a subclass, the programmer must override that member in the subclass.

- In a “has a” relationship, a class object has a reference to an object of another class as a member. In an “is a” relationship, an object of a subclass type may also be treated as an object of the superclass type. “Is a” is inheritance. “Has a” is composition.

- A reference to a subclass object may be converted implicitly to a reference for a superclass object.

- It is possible to convert a superclass reference to a subclass reference by using an explicit cast. If the target object is not a subclass object, a ClassCastException is thrown.

- A superclass specifies commonality. All classes derived from a superclass inherit the capabilities of that superclass. In the object-oriented design process, the designer looks for commonality among classes and factors it out to form superclasses. Then, subclasses are customized beyond the capabilities inherited from the superclass.
• Reading a set of subclass declarations can be confusing, because inherited superclass members are not listed in the subclass declarations, but these members are indeed present in the subclasses.

• With polymorphism, it becomes possible to design and implement systems that are more extensible. Programs can be written to process objects of types that may not exist when the program is under development.

• Polymorphic programming can eliminate the need for `switch` logic, thus avoiding the kinds of errors associated with `switch` logic.

• An abstract method is declared by preceding the method’s definition with the keyword `abstract` in the superclass.

• There are many situations in which it is useful to define classes for which the programmer never intends to instantiate any objects. Such classes are called abstract classes. Because these are used only as superclasses, typically they are called abstract superclasses. A program cannot instantiate objects of an abstract class.

• Classes from which a program can instantiate objects are called concrete classes.

• A class is made abstract by declaring it with the keyword `abstract`.

• If a subclass is derived from a superclass with an abstract method without supplying a definition for that abstract method in the subclass, that method remains abstract in the subclass. Consequently, the subclass is also an abstract class.

• When a request is made through a superclass reference to use a method, Java chooses the correct overridden method in the subclass associated with the object.

• Through the use of polymorphism, one method call can cause different actions to occur, depending on the type of the object receiving the call.

• Although we cannot instantiate objects of abstract superclasses, we can declare references to abstract superclasses. Such references can then be used to enable polymorphic manipulations of subclass objects when such objects are instantiated from concrete classes.

• New classes are regularly added to systems. New classes are accommodated by dynamic method binding (also called late binding). The type of an object need not be known at compile time for a method call to be compiled. At execution time, the appropriate method of the receiving object is selected.

• With dynamic method binding, at execution time the call to a method is routed to the method version appropriate for the class of the object receiving the call.

• When a superclass provides a method, subclasses can override the method, but they do not have to override it. Thus a subclass can use a superclass’s version of a method.

• An interface definition begins with the keyword `interface` and contains a set of public abstract methods. Interfaces may also contain public static final data.

• To use an interface, a class must specify that it implements the interface and that class must define every method in the interface with the number of arguments and the return type specified in the interface definition.

• An interface is typically used in place of an abstract class when there is no default implementation to inherit.

• When a class implements an interface, the same “is a” relationship provided by inheritance applies.

• To implement more than one interface, simply provide a comma-separated list of interface names after keyword `implements` in the class definition.

• Inner classes are defined inside the scope of other classes.
• An inner class can also be defined inside a method of a class. Such an inner class has access to its outer class’s members and to the \texttt{final} local variables for the method in which it is defined.

• Inner class definitions are used mainly in event handling.

• Class \texttt{JFrame} provides the basic attributes and behaviors of a window—a title bar and buttons to minimize, maximize and close the window.

• An inner class object has access to all the variables and methods of the outer class object.

• Because an anonymous inner class has no name, one object of the anonymous inner class must be created at the point where the class is defined in the program.

• An anonymous inner class can implement an interface or extend a class.

• The event generated when the user clicks the window’s close box is a window closing event.

• Method \texttt{addWindowListener} registers a window event listener. The argument to \texttt{addWindowListener} must be a reference to an object that is a \texttt{WindowListener} (package \texttt{java.awt.event}).

• For event handling interfaces with more than one method, Java provides a corresponding class (called an adapter class) that already implements all the methods in the interface for you. Class \texttt{WindowAdapter} implements interface \texttt{WindowListener}, so every \texttt{WindowAdapter} object is a \texttt{WindowListener}.

• Compiling a class that contains inner classes results in a separate \texttt{.class} file for every class.

• Inner classes with class names can be defined as \texttt{public}, \texttt{protected}, package access or \texttt{private} and are subject to the same usage restrictions as other members of a class.

• To access the outer class’s \texttt{this} reference, use \texttt{OuterClassName.this}.

• The outer class is responsible for creating objects of its non\texttt{static} inner classes.

• An inner class can be declared \texttt{static}.

\textbf{TERMINOLOGY}

\begin{itemize}
  \item \texttt{abstract} class
  \item \texttt{abstract} method
  \item \texttt{abstract} superclass
  \item abstraction
  \item anonymous inner class
  \item base class
  \item \texttt{Boolean} class
  \item \texttt{Character} class
  \item class hierarchy
  \item client of a class
  \item composition
  \item direct superclass
  \item \texttt{Double} class
  \item dynamic method binding
  \item extends
  \item extensibility
  \item \texttt{final} class
  \item \texttt{final} instance variable
  \item \texttt{final} method
  \item garbage collection
  \item “has a” relationship
  \item hierarchical relationship
  \item implementation inheritance
  \item implicit reference conversion
  \item indirect superclass
  \item infinite recursion error
  \item inheritance
  \item inheritance hierarchy
  \item inner class
  \item \texttt{Integer} class
  \item interface
  \item interface inheritance
  \item “is a” relationship
  \item \texttt{JFrame} class
  \item late binding
  \item \texttt{Long} class
  \item member access control
  \item member object
  \item method overriding
  \item multiple inheritance
  \item \texttt{Number} class
  \item \texttt{Object} class
\end{itemize}
SELF-REVIEW EXERCISES

9.1 Fill in the blanks in each of the following statements:
   a) If the class Alpha inherits from the class Beta, class Alpha is called the ________
      class and class Beta is called the ________ class.
   b) Inheritance enables ________, which saves time in development and encourages using
      previously proven and high-quality software components.
   c) An object of a ________ class can be treated as an object of its corresponding
      ________ class.
   d) The four member access specifiers are ________, ________, ________ and ________.
   e) A “has a” relationship between classes represents ________ and an “is a” relationship
      between classes represents ________.
   f) Using polymorphism helps eliminate ________ logic.
   g) If a class contains one or more abstract methods, it is an ________ class.
   h) A method call resolved at run-time is referred to as ________ binding.
   i) A subclass may call any non-private superclass method by prepending ________ to
      the method call.

9.2 State whether each of the following statements is true or false. If false, explain why.
   a) A superclass typically represents a larger number of objects than its subclass represents
      (true/false).
   b) A subclass typically encapsulates less functionality than does its superclass. (true/false).

ANSWERS TO SELF-REVIEW EXERCISES

9.1 a) sub, super. b) software reusability. c) sub, super. d) public, protected, private and
      package access. e) composition, inheritance. f) switch. g) abstract. h) dynamic. i) super.

9.2 a) True.
      b) False. A subclass includes all the functionality of its superclass.

EXERCISES

9.3 Consider the class Bicycle. Given your knowledge of some common components of bi-
      cycles, show a class hierarchy in which the class Bicycle inherits from other classes, which, in turn,
      inherit from yet other classes. Discuss the instantiation of various objects of class Bicycle. Discuss
      inheritance from class Bicycle for other closely related subclasses.
9.4 Define each of the following terms: single inheritance, multiple inheritance, interface, superclass and subclass.

9.5 Discuss why casting a superclass reference to a subclass reference is potentially dangerous.

9.6 Distinguish between single inheritance and multiple inheritance. What feature of Java helps realize the benefits of multiple inheritance?

9.7 (True/False) A subclass is generally smaller than its superclass.

9.8 (True/False) A subclass object is also an object of that subclass’s superclass.

9.9 Some programmers prefer not to use protected access because it breaks information hiding in the superclass. Discuss the relative merits of using protected access vs. private access in superclasses.

9.10 Many programs written with inheritance could be solved with composition instead, and vice versa. Discuss the relative merits of these approaches in the context of the Point, Circle, Cylinder class hierarchy in this chapter. Rewrite the program of Fig. 9.22–Fig. 9.26 (and the supporting classes) to use composition rather than inheritance. After you do this, reassess the relative merits of the two approaches both for the Point, Circle, Cylinder problem and for object-oriented programs in general.

9.11 Rewrite the Point, Circle, Cylinder program of Fig. 9.22–Fig. 9.26 as a Point, Square, Cube program. Do this two ways—once with inheritance and once with composition.

9.12 In the chapter, we stated, “When a superclass method is inappropriate for a subclass, that method can be overridden in the subclass with an appropriate implementation.” If this is done, does the subclass-is-a-superclass-object relationship still hold? Explain your answer.

9.13 Study the inheritance hierarchy of Fig. 9.2. For each class, indicate some common attributes and behaviors consistent with the hierarchy. Add some other classes (e.g., UndergraduateStudent, GraduateStudent, Freshman, Sophomore, Junior, Senior), to enrich the hierarchy.

9.14 Write an inheritance hierarchy for classes Quadrilateral, Trapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the superclass of the hierarchy. Make the hierarchy as deep (i.e., as many levels) as possible. The private data of Quadrilateral should include the (x, y) coordinate pairs for the four endpoints of the Quadrilateral. Write a driver program that instantiates and displays objects of each of these classes. [In Chapter 11, “Graphics and Java2D,” you will learn how to use Java’s drawing capabilities.]

9.15 Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form those shapes into a shape hierarchy. Your hierarchy should have superclass Shape, from which class TwoDimensionalShape and class ThreeDimensionalShape are derived. Once you have developed the hierarchy, define each of the classes in the hierarchy. We will use this hierarchy in the exercises to process all shapes as objects of superclass Shape.

9.16 How is it that polymorphism enables you to program “in the general” rather than “in the specific”? Discuss the key advantages of programming “in the general.”

9.17 Discuss the problems of programming with switch logic. Explain why polymorphism is an effective alternative to using switch logic.

9.18 Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?

9.19 Distinguish between nonabstract methods and abstract methods.

9.20 (True/False) All methods in an abstract superclass must be declared abstract.

9.21 Suggest one or more levels of abstract superclasses for the Shape hierarchy discussed in the beginning of this chapter (the first level is Shape and the second level consists of the classes TwoDimensionalShape and ThreeDimensionalShape).
9.22 How does polymorphism promote extensibility?

9.23 You have been asked to develop a flight simulator that will have elaborate graphical outputs. Explain why polymorphic programming would be especially effective for a problem of this nature.

9.24 Develop a basic graphics package. Use the Shape class inheritance hierarchy from Figure 9.3. Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles. Interact with the user. Let the user specify the position, size, shape and fill colors to be used in drawing each shape. The user can specify many items of the same shape. As you create each shape, place a Shape reference to each new Shape object into an array. Each class has its own draw method. Write a polymorphic screen manager that walks through the array sending draw messages to each object in the array to form a screen image. Redraw the screen image each time the user specifies an additional shape. Investigate the methods of class Graphics to help draw each shape.

9.25 Modify the payroll system of Fig. 9.16–Fig. 9.21 to add private instance variables birthDate (use class Date from Figure 8.13) and departmentCode (an int) to class Employee. Assume this payroll is processed once per month. Then, as your program calculates the payroll for each Employee (polymorphically), add a $100.00 bonus to the person’s payroll amount if this is the month in which the Employee’s birthday occurs.

9.26 In Exercise 9.15, you developed a Shape class hierarchy and defined the classes in the hierarchy. Modify the hierarchy so that class Shape is an abstract superclass containing the interface to the hierarchy. Derive TwoDimensionalShape and ThreeDimensionalShape from class Shape—these classes should also be abstract. Use an abstract print method to output the type and dimensions of each class. Also include area and volume methods so these calculations can be performed for objects of each concrete class in the hierarchy. Write a driver program that tests the Shape class hierarchy.

9.27 Rewrite your solution to Exercise 9.26 to use a Shape interface instead of an abstract Shape class.

9.28 (Drawing Application) Modify the drawing program of Exercise 8.19 to create a drawing application that draws random lines, rectangles and ovals. [Note: Like an applet, a JFrame has a paint method that you can override to draw on the background of the JFrame.]

For this exercise, modify the MyLine, MyOval and MyRect classes of Exercise 8.19 to create the class hierarchy in Fig. 9.45. The classes of the MyShape hierarchy should be “smart” shape classes where objects of these classes know how to draw themselves (if provided with a Graphics object that tells them where to draw). The only switch or if/else logic in this program should be to determine the type of shape object to create (use random numbers to pick the shape type and the coordinates of each shape). Once an object from this hierarchy is created, it will be manipulated for the rest of its lifetime as a superclass MyShape reference.

![MyShape hierarchy](image-url)

Fig. 9.45 The MyShape hierarchy.
Class **MyShape** in Fig. 9.45 must be **abstract**. The only data representing the coordinates of the shapes in the hierarchy should be defined in class **MyShape**. Lines, rectangles and ovals can all be drawn if you know two points in space. Lines require \( x_1, y_1, x_2 \) and \( y_2 \) coordinates. The **drawLine** method of the **Graphics** class will connect the two points supplied with a line. If you have the same four coordinate values \( (x_1, y_1, x_2, y_2) \) for ovals and rectangles, you can calculate the four arguments needed to draw them. Each requires an upper-left \( x \)-coordinate value (minimum of the two \( x \)-coordinate values), an upper-left \( y \)-coordinate value (minimum of the two \( y \) coordinate values), a \( width \) (difference between the two \( x \)-coordinate values; must be nonnegative) and a \( height \) (difference between the two \( y \)-coordinate values; must be nonnegative). [Note: In Chapter 12, each \( x,y \) pair will be captured by using mouse events from mouse interactions between the user and the program’s background. These coordinates will be stored in an appropriate shape object as selected by the user. As you begin the exercise, you will use random coordinate values as arguments to the constructor.]

In addition to the data for the hierarchy, class **MyShape** should define at least the following methods:

a) A constructor with no arguments that sets the coordinates to 0.

b) A constructor with arguments that sets the coordinates to the supplied values.

c) Set methods for each individual piece of data that allow the programmer to independently set any piece of data for a shape in the hierarchy (e.g., if you have an instance variable \( x_1 \), you should have a method **setX1**).

d) Get methods for each individual piece of data that allow the programmer to independently retrieve any piece of data for a shape in the hierarchy (e.g., if you have an instance variable \( x_1 \), you should have a method **getX1**).

e) The **abstract** method

\[
\text{public abstract void draw( Graphics g );}
\]

This method will be called from the program’s **paint** method to draw a shape onto the screen.

The preceding methods are required. If you would like to provide more methods for flexibility, please do so. However, be sure that any method you define in this class is a method that would be used by all shapes in the hierarchy.

All data must be **private** to class **MyShape** in this exercise (this forces you to use proper encapsulation of the data and provide proper set/get methods to manipulate the data). You are not allowed to define new data that can be derived from existing information. As explained previously, the upper-left \( x \), upper-left \( y \), \( width \) and \( height \) needed to draw an oval or rectangle can be calculated if you already know two points in space. All subclasses of **MyShape** should provide two constructors that mimic those provided by class **MyShape**.

Objects of the **MyOval** and **MyRect** classes should not calculate their upper-left \( x \)-coordinate, upper-left \( y \)-coordinate, \( width \) and \( height \) until they are about to draw. Never modify the \( x_1, y_1, x_2 \) and \( y_2 \) coordinates of a **MyOval** or **MyRect** object to prepare to draw them. Instead, use the temporary results of the calculations described above. This will help us enhance the program in Chapter 12 by allowing the user to select each shape’s coordinates with the mouse.

There should be no **MyLine**, **MyOval** or **MyRect** references in the program—only **MyShape** references that refer to **MyLine**, **MyOval** and **MyRect** objects are allowed. The program should keep an array of **MyShape** references containing all shapes. The program’s **paint** method should walk through the array of **MyShape** references and draw every shape (i.e., call every shape’s **draw** method).

Begin by defining class **MyShape**, class **MyLine** and an application to test your classes. The application should have a **MyShape** instance variable that can refer to one **MyLine** object (created
in the application’s constructor). The `paint` method (for your subclass of `JFrame`) should draw the shape with a statement like

```java
currentShape.draw( g );
```

where `currentShape` is the `MyShape` reference and `g` is the `Graphics` object that the shape will use to draw itself on the background of the window.

9.29 Next, change the single `MyShape` reference into an array of `MyShape` references and hard code several `MyLine` objects into the program for drawing. The application’s `paint` method should walk through the array of shapes and draw every shape.

After the preceding part is working, you should define the `MyOval` and `MyRect` classes and add objects of these classes into the existing array. For now, all the shape objects should be created in the constructor for your subclass of `JFrame`. In Chapter 12, we will create the objects when the user chooses a shape and begins drawing it with the mouse.

In Exercise 9.28, you defined a `MyShape` hierarchy in which classes `MyLine`, `MyOval` and `MyRect` subclass `MyShape` directly. If the hierarchy was properly designed, you should be able to see the tremendous similarities between the `MyOval` and `MyRect` classes. Redesign and reimplement the code for the `MyOval` and `MyRect` classes to “factor out” the common features into the abstract class `MyBoundedShape` to produce the hierarchy in Fig. 9.46.

Class `MyBoundedShape` should define two constructors that mimic the constructors of class `MyShape` and methods that calculate the upper-left x-coordinate, upper-left y-coordinate, width and height. No new data pertaining to the dimensions of the shapes should be defined in this class. Remember, the values needed to draw an oval or rectangle can be calculated from two \((x,y)\) coordinates. If designed properly, the new `MyOval` and `MyRect` classes should each have two constructors and a `draw` method.

![MyShape hierarchy diagram](image)

**Fig. 9.46** The `MyShape` hierarchy.
Objectives

• To be able to create and manipulate nonmodifiable character string objects of class `String`.
• To be able to create and manipulate modifiable character string objects of class `StringBuffer`.
• To be able to create and manipulate objects of class `Character`.
• To be able to use a `StringTokenizer` object to break a `String` object into individual components called tokens.

The chief defect of Henry King
Was chewing little bits of string.
Hilaire Belloc

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences.
William Strunk, Jr.

I have made this letter longer than usual, because I lack the time to make it short.
Blaise Pascal

The difference between the almost-right word & the right word is really a large matter—it’s the difference between the lightning bug and the lightning.
Mark Twain

Mum’s the word.
Miguel de Cervantes, Don Quixote de la Mancha
10.1 Introduction

In this chapter, we introduce Java’s string and character-processing capabilities. The techniques discussed here are appropriate for validating program input, displaying information to users and other text-based manipulations. The techniques also are appropriate for developing text editors, word processors, page-layout software, computerized typesetting systems and other kinds of text-processing software. We have already presented several string-processing capabilities in the text. This chapter discusses in detail the capabilities of class \texttt{String}, class \texttt{StringBuffer} and class \texttt{Character} from the \texttt{java.lang} package and class \texttt{StringTokenizer} from the \texttt{java.util} package. These classes provide the foundation for string and character manipulation in Java.
10.2 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of Java source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the computer as a series of instructions used to accomplish a task. A program might contain character constants. A character constant is an integer value represented as a character in single quotes. As we stated previously, the value of a character constant is the integer value of the character in the Unicode character set. For example, 'z' represents the integer value of z, and '
' represents the integer value of newline. See Appendix D for the integer equivalents of these characters.

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters, such as +, -, *, /, $ and others. A string is an object of class String. String literals or string constants (often called anonymous String objects) are written as a sequence of characters in double quotation marks as follows:

- "John Q. Doe" (a name)
- "9999 Main Street" (a street address)
- "Waltham, Massachusetts" (a city and state)
- "(201) 555-1212" (a telephone number)

A String may be assigned in a declaration to a String reference. The declaration

```java
String color = "blue";
```

initializes String reference color to refer to the anonymous String object "blue".

**Performance Tip 10.1**

Java treats all anonymous Strings with the same contents as one anonymous String object that has many references. This conserves memory.

10.3 String Constructors

Class String provides nine constructors for initializing String objects in a variety of ways. Seven of the constructors are demonstrated in Fig. 10.1. All the constructors are used in the StringConstructors application’s main method.

Line 25 instantiates a new String object and assigns it to reference s1, using class String’s default constructor. The new String object contains no characters (the empty string) and has a length of 0.

Line 26 instantiates a new String object and assigns it to reference s2, using class String’s copy constructor. The new String object contains a copy of the characters in the String object s that is passed as an argument to the constructor.

```java
1 // Fig. 10.1: StringConstructors.java
2 // This program demonstrates the String class constructors.
3
4 // Java extension packages
5 import javax.swing.*;
6
Fig. 10.1 Demonstrating the String class constructors (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
public class StringConstructors {

    // test String constructors
    public static void main( String args[] )
    {
        char charArray[] = { 'b', 'i', 'r', 't', 'h', ' ',
                             'd', 'a', 'y' };
        byte byteArray[] = { ( byte ) 'n', ( byte ) 'e',
                             ( byte ) 'w', ( byte ) 'y',
                             ( byte ) 'e', ( byte ) 'a', ( byte ) 'r' };

        StringBuffer buffer;

        String s, s1, s2, s3, s4, s5, s6, output;

        s = new String( "hello" );
        buffer = new StringBuffer( "Welcome to Java Programming!" );

        // use String constructors
        s1 = new String();
        s2 = new String( s );
        s3 = new String( charArray );
        s4 = new String( charArray, 6, 3 );
        s5 = new String( byteArray, 4, 4 );
        s6 = new String( byteArray );
        s7 = new String( buffer );

        // append Strings to output
        output = "s1 = " + s1 + "\ns2 = " + s2 + "\ns3 = " + s3 + "\n" + s4 + "\n" + s5 + "\n" + s6 + "\n" + s7;

        JOptionPane.showMessageDialog( null, output,
                                        "Demonstrating String Class Constructors",
                                        JOptionPane.INFORMATION_MESSAGE);

        System.exit( 0 );
    }
}

Fig. 10.1 Demonstrating the String class constructors (part 2 of 2).
Software Engineering Observation 10.1

In most cases, it is not necessary to make a copy of an existing `String` object. `String` objects are immutable—their character contents cannot be changed after they are created. Also, if there are one or more references to a `String` object (or any object for that matter), the object cannot be reclaimed by the garbage collector. Thus, a `String` reference cannot be used to modify a `String` object or to delete a `String` object from memory as in other programming languages, such as C or C++.

Line 27 instantiates a new `String` object and assigns it to reference `s3` using class `String`’s constructor that takes a character array as an argument. The new `String` object contains a copy of the characters in the array.

Line 28 instantiates a new `String` object and assigns it to reference `s4` using class `String`’s constructor that takes a `char` array and two integers as arguments. The second argument specifies the starting position (the `offset`) from which characters in the array are copied. The third argument specifies the number of characters (the `count`) to be copied from the array. The new `String` object contains a copy of the specified characters in the array. If the `offset` or the `count` specified as arguments result in accessing an element outside the bounds of the character array, a `StringIndexOutOfBoundsException` is thrown. We discuss exceptions in detail in Chapter 14.

Line 29 instantiates a new `String` object and assigns it to reference `s5` using class `String`’s constructor that receives a `byte` array and two integers as arguments. The second and third arguments specify the `offset` and `count`, respectively. The new `String` object contains a copy of the specified `bytes` in the array. If the `offset` or the `count` specified as arguments result in accessing an element outside the bounds of the character array, a `StringIndexOutOfBoundsException` is thrown.

Line 30 instantiates a new `String` object and assigns it to reference `s6` using class `String`’s constructor that takes a `byte` array as an argument. The new `String` object contains a copy of the bytes in the array.

Line 31 instantiates a new `String` object and assigns it to reference `s7` using class `String`’s constructor that receives a `StringBuffer` as an argument. A `StringBuffer` is a dynamically resizable and modifiable string. The new `String` object contains a copy of the characters in the `StringBuffer`. Line 22 creates a new object of class `StringBuffer` using the constructor that receives a `String` argument (in this case "Welcome to Java Programming") and assign the new object to reference `buffer`. We discuss `StringBuffers` in detail later in this chapter. The screen capture for the program displays the contents of each `String`.

10.4 String Methods `length`, `charAt` and `getChars`

The application of Fig. 10.2 presents `String` methods `length`, `charAt` and `getChars`, which determine the length of a `String`, get the character at a specific location in a `String` and get the entire set of characters in a `String`, respectively.

Line 28 uses `String` method `length` to determine the number of characters in `String s1`. Like arrays, `Strings` always know their own size. However, unlike arrays, `Strings` do not have a `length` instance variable that specifies the number of elements in a `String`.
// Fig. 10.2: StringMiscellaneous.java
// This program demonstrates the length, charAt and getChars
// methods of the String class.
//
// Note: Method getChars requires a starting point
// and ending point in the String. The starting point is the
// actual subscript from which copying starts. The ending point
// is one past the subscript at which the copying ends.

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

public class StringMiscellaneous {
    // test miscellaneous String methods
    public static void main( String args[] ) {
        String s1, output;
        char charArray[];

        s1 = new String( "hello there" );
        charArray = new char[ 5 ];

        // output the string
        output = "s1: " + s1;

        // test length method
        output += "\nLength of s1: " + s1.length();

        // loop through characters in s1 and display reversed
        output += "\nThe string reversed is: ";
        for ( int count = s1.length() - 1; count >= 0; count-- )
            output += s1.charAt( count ) + " ";

        // copy characters from string into char array
        s1.getChars( 0, 5, charArray, 0 );
        output += "\nThe character array is: ";

        for ( int count = 0; count < charArray.length; count++ )
            output += charArray[ count ];

        JOptionPane.showMessageDialog( null, output,
            "Demonstrating String Class Constructors",
            JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }
}

Fig. 10.2 The String class character manipulation methods (part 1 of 2).
Common Programming Error 10.1

Attempting to determine the length of a `String` via an instance variable called `length` (e.g., `s1.length`) is a syntax error. The `String` method `length` must be used. (e.g., `s1.length()`).

The `for` structure at lines 33–34 appends to `output` the characters of the `String s1` in reverse order. The `String` method `charAt` returns the character at a specific position in the `String`. Method `charAt` receives an integer argument that is used as the position number (or index) and returns the character at that position. Like arrays, the first element of a `String` is considered to be at position 0.

Common Programming Error 10.2

Attempting to access a character that is outside the bounds of a `String` (i.e., an index less than 0 or an index greater than or equal to the `String`’s length) results in a `StringIndexOutOfBoundsException`.

Line 37 uses `String` method `getChars` to copy the characters of a `String` into a character array. The first argument is the starting index from which characters are copied in the `String`. The second argument is the index that is one past the last character to be copied from the `String`. The third argument is the character array into which the characters are copied. The last argument is the starting index where the copied characters are placed in the character array. Next, the `char` array contents are appended one character at a time to `String output` with the `for` structure at lines 40–41 for display purposes.

10.5 Comparing Strings

Java provides a variety of methods for comparing `String` objects; these are demonstrated in the next two examples. To understand just what it means for one string to be “greater than” or “less than” another string, consider the process of alphabetizing a series of last names. The reader would, no doubt, place “Jones” before “Smith” because the first letter of “Jones” comes before the first letter of “Smith” in the alphabet. But the alphabet is more than just a list of 26 letters—it is an ordered list of characters. Each letter occurs in a specific position within the list. “Z” is more than just a letter of the alphabet; “Z” is specifically the twenty-sixth letter of the alphabet.

How does the computer know that one letter comes before another? All characters are represented inside the computer as numeric codes (see Appendix D). When the computer compares two strings, it actually compares the numeric codes of the characters in the strings.
Figure 10.3 demonstrates the **String** methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches` and demonstrates using the equality operator `==` to compare **String** objects.

```java
// Fig. 10.3: StringCompare.java
// This program demonstrates the methods equals,
// equalsIgnoreCase, compareTo, and regionMatches
// of the String class.

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

public class StringCompare {

    public static void main( String args[] ) {
        String s1, s2, s3, s4, output;

        s1 = new String( "hello" );
        s2 = new String( "good bye" );
        s3 = new String( "Happy Birthday" );
        s4 = new String( "happy birthday" );

        output = "s1 = " + s1 + "\n" + s2 + "\n" + s3 + "\n" + s4 + "\n\n";

        // test for equality
        if ( s1.equals( "hello" ) )
            output += "s1 equals " + s1 + "\n";
        else
            output += "s1 does not equal " + s1 + "\n";

        // test for equality with ==
        if ( s1 == "hello" )
            output += "s1 equals \"hello\"\n";
        else
            output += "s1 does not equal \"hello\"\n";

        // test for equality (ignore case)
        if ( s3.equalsIgnoreCase( s4 ) )
            output += "s3 equals s4\n";
        else
            output += "s3 does not equal s4\n";

        // test compareTo
        output += "\n" + s1.compareTo( s2 ) + "\n" + s2.compareTo( s1 ) + "\n" + s1.compareTo( s4 ) + "\n" + s2.compareTo( s4 ) + "\n" + s3.compareTo( s4 ) + "\n" + s3.compareTo( s3 ) + "\n";
    }
}
```

**Fig. 10.3** Demonstrating **String** comparisons (part 1 of 2).
The condition in the `if` structure at line 25 uses method `equals` to compare `String s1` and anonymous `String "hello"` for equality. Method `equals` (inherited into `String` from its superclass `Object`) tests any two objects for equality—the contents of the two objects are identical. The method returns `true` if the objects are equal and `false` otherwise. The preceding condition is `true` because `String s1` was initialized with a copy of the anonymous `String "hello"`. Method `equals` uses a lexicographical comparison—the integer Unicode values that represent each character in each `String` are
compared. Thus, if the String "hello" is compared with the String "HELLO", the result is false, because the integer representation of a lowercase letter is different from that of the corresponding uppercase letter.

The condition in the if structure at line 31 uses the equality operator == to compare String s1 for equality with the anonymous String "hello". Operator == has different functionality when it is used to compare references and when it is used to compare values of primitive data types. When primitive data type values are compared with ==, the result is true if both values are identical. When references are compared with ==, the result is true if both references refer to the same object in memory. To compare the actual contents (or state information) of objects for equality, methods (such as equals) must be invoked. The preceding condition evaluates to false in this program because the reference s1 was initialized with the statement

```java
s1 = new String( "hello" );
```

which creates a new String object with a copy of anonymous String "hello" and assigns the new object to reference s1. If s1 had been initialized with the statement

```java
s1 = "hello";
```

which directly assigns the anonymous String "hello" to the reference s1, the condition would be true. Remember that Java treats all anonymous String objects with the same contents as one anonymous String object that has many references. Thus, lines 16, 25 and 31 all refer to the same anonymous String object "hello" in memory.

### Common Programming Error 10.3

Comparing references with == can lead to logic errors, because == compares the references to determine whether they refer to the same object, not whether two objects have the same contents. When two identical (but separate) objects are compared with ==, the result will be false. When comparing objects to determine whether they have the same contents, use method equals.

If you are sorting Strings, you may compare them for equality with method equalsIgnoreCase, which ignores the case of the letters in each String when performing the comparison. Thus, the String "hello" and the String "HELLO" compare as equal. The if structure at line 37 uses String method equalsIgnoreCase to compare String s3—Happy Birthday—for equality with String s4—happy birthday. The result of this comparison is true, because the comparison ignores case sensitivity.

Lines 44–48 use String method compareTo to compare String objects. For example, line 44 compares String s1 to String s2. Method compareTo returns 0 if the Strings are equal, a negative number if the String that invokes compareTo is less than the String that is passed as an argument and a positive number if the String that invokes compareTo is greater than the String that is passed as an argument. Method compareTo uses a lexicographical comparison—it compares the numeric values of corresponding characters in each String.

The condition in the if structure at line 52 uses String method regionMatches to compare portions of two String objects for equality. The first argument is the starting index in the String that invokes the method. The second argument is a comparison
The third argument is the starting index in the comparison method. The last argument is the number of characters to compare between the two strings. The method returns true only if the specified number of characters are lexicographically equal.

Finally, the condition in the if structure at line 59 uses a second version of the String method regionMatches to compare portions of two String objects for equality. If the first argument is true, the method ignores the case of the characters being compared. The remaining arguments are identical to those described for the for-argument regionMatches method.

The second example of this section (Fig. 10.4) demonstrates the startsWith and endsWith methods of class String. Application StringStartEnd's main method defines an array of Strings called strings containing "started", "starting", "ended", and "ending". The remainder of method main consists of three for structures that test the elements of the array to determine whether they start with or end with a particular set of characters.

```java
// Fig. 10.4: StringStartEnd.java
// This program demonstrates the methods startsWith and
// endsWith of the String class.

import javax.swing.*;

public class StringStartEnd {

    public static void main( String args[] )
    {
        String strings[] =
            { "started", "starting", "ended", "ending" };
        String output = "";

        // test method startsWith
        for ( int count = 0; count < strings.length; count++ )
            if ( strings[ count ].startsWith( "st" ) )
                output += "" + strings[ count ] + " starts with \"st\"\n";

        output += "\n";

        // test method startsWith starting from position
        // 2 of the string
        for ( int count = 0; count < strings.length; count++ )
            if ( strings[ count ].startsWith( "art", 2 ) )
                output += "" + strings[ count ] + " starts with \"art\" at position 2\n";

        System.out.println( output );
    }
}
```

Fig. 10.4  String class startsWith and endsWith methods (part 1 of 2).
The first for structure (lines 19–23) uses the version of method `startsWith` that takes a `String` argument. The condition in the if structure (line 21) determines whether the `String` at location `count` of the array starts with the characters "st". If so, the method returns `true` and the program appends `strings[count]` to `output` for display purposes.

The second for structure (lines 29–33) uses the version of method `startsWith` that takes a `String` and an integer as arguments. The integer argument specifies the index at which the comparison should begin in the `String`. The condition in the if structure (line 31) determines whether the `String` at location `count` of the array starts with the characters "art" beginning with the character at index 2 in each `String`. If so, the method returns `true` and the program appends `strings[count]` to `output` for display purposes.

The third for structure (line 38–42) uses method `endsWith`, which takes a `String` argument. The condition in the if structure (40) determines whether the `String` at location `count` of the array ends with the characters "ed". If so, the method returns `true` and the program appends `strings[count]` to `output` for display purposes.

### 10.6 String Method `hashCode`

Often, it is necessary to store `Strings` and other data types in a manner that allows the information to be found quickly. One of the best ways to store information for fast lookup
is a hash table. A hash table stores information using a special calculation on the object to be stored that produces a hash code. The hash code is used to choose the location in the table at which to store the object. When the information needs to be retrieved, the same calculation is performed, the hash code is determined and a lookup of that location in the table results in the value that was stored there previously. Every object has the ability to be stored in a hash table. Class Object defines method `hashCode` to perform the hash code calculation. This method is inherited by all subclasses of Object. Method `hashCode` is overridden by String to provide a good hash code distribution based on the contents of the String. We will say more about hashing in Chapter 20.

The example in Fig. 10.5 demonstrates the `hashCode` method for two `String` objects containing "hello" and "Hello". Note that the hash code value for each `String` is different. That is because the Strings themselves are lexicographically different.

```java
// Fig. 10.5: StringHashCode.java
// This program demonstrates the method
// hashCode of the String class.

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

public class StringHashCode {

    // test String hashCode method
    public static void main( String args[] )
    {
        String s1 = "hello", s2 = "Hello";

        String output = "The hash code for " + s1 + " is " + s1.hashCode() + "\nThe hash code for " + s2 + " is " + s2.hashCode();

        JOptionPane.showMessageDialog( null, output,
        "Demonstrating String Method hashCode",
        JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }

} // end class StringHashCode
```

Fig. 10.5 `String` class `hashCode` method.
10.7 Locating Characters and Substrings in Strings

Often it is useful to search for a character or set of characters in a String. For example, if you are creating your own word processor, you might want to provide a capability for searching through the document. The application of Fig. 10.6 demonstrates the many versions of String methods indexOf and lastIndexOf that search for a specified character or substring in a String. All the searches in this example are performed on the String letters (initialized with "abcdefghijklmabcdefghijklm") in method main of class StringIndexMethods.

```java
// Fig. 10.6: StringIndexMethods.java
// This program demonstrates the String class index methods.

import javax.swing.*;

public class StringIndexMethods {
  public static void main( String args[] )
  {
    String letters = "abcdefghijklmnopqrstuvwxyzabcdeghijklm";

    // test indexOf to locate a character in a string
    String output = "'c' is located at index "+
    letters.indexOf( 'c' );
    output += "
    output += "'a' is located at index " +
    letters.indexOf( 'a', 1 );
    output += "$" is located at index " +
    letters.indexOf( '$' );

    // test lastIndexOf to find a character in a string
    output += "Last 'c' is located at index "+
    letters.lastIndexOf( 'c' );
    output += "Last 'a' is located at index " +
    letters.lastIndexOf( 'a', 25 );
    output += "Last '$' is located at index " +
    letters.lastIndexOf( '$' );

    // test indexOf to locate a substring in a string
    output += "'def' is located at index " +
    letters.indexOf( "def" );
    output += "'def' is located at index " +
    letters.indexOf( "def", 7 );
  }
}
```

Fig. 10.6 The String class searching methods (part 1 of 2).
The statements at lines 16–23 use method `indexOf` to locate the first occurrence of a character in a `String`. If `indexOf` finds the character, `indexOf` returns the index of that character in the `String`; otherwise, `indexOf` returns -1. There are two versions of `indexOf` that search for characters in a `String`. The expression on line 17 uses method `indexOf` that takes one integer argument, which is the integer representation of a character. Remember that a character constant in single quotes is of type `char` and specifies the integer representation of the character in the Unicode character set. The expression at line 20 uses the second version of method `indexOf`, which takes two integer arguments—the integer representation of a character and the starting index at which the search of the `String` should begin.
The statements at lines 26–33 use method `lastIndexOf` to locate the last occurrence of a character in a `String`. Method `lastIndexOf` performs the search from the end of the `String` toward the beginning of the `String`. If method `lastIndexOf` finds the character, `lastIndexOf` returns the index of that character in the `String`; otherwise, `lastIndexOf` returns –1. There are two versions of `lastIndexOf` that search for characters in a `String`. The expression at line 27 uses the version of method `lastIndexOf` that takes one integer argument that is the integer representation of a character. The expression at line 30 uses the version of method `lastIndexOf` that takes two integer arguments—the integer representation of a character and the highest index from which to begin searching backward for the character.

Lines 36–53 of the program demonstrate versions of methods `indexOf` and `lastIndexOf` that each take a `String` as the first argument. These versions of the methods perform identically to those described above except that they search for sequences of characters (or substrings) that are specified by their `String` arguments.

### 10.8 Extracting Substrings from Strings

Class `String` provides two `substring` methods to enable a new `String` object to be created by copying part of an existing `String` object. Each method returns a new `String` object. Both methods are demonstrated in Fig. 10.7.

```java
// Fig. 10.7: SubString.java
// This program demonstrates the String class substring methods.
// Java extension packages
import javax.swing.*;

public class SubString {

    // test String substring methods
    public static void main( String args[] )
    {
        String letters = "abcdefghijklmabcdefghijklm";

        // test substring methods
        String output = "Substring from index 20 to end is " +
                        "\"" + letters.substring( 20 ) + "\"\n";

        output += "Substring from index 0 up to 6 is " +
                  "\"" + letters.substring( 0, 6 ) + "\"";

        JOptionPane.showMessageDialog( null, output,
                                        "Demonstrating String Class Substring Methods",
                                        JOptionPane.INFORMATION_MESSAGE);

        System.exit( 0 );
    }
}
```

Fig. 10.7  `String` class `substring` methods (part 1 of 2).
The expression `letters.substring(20)` from line 17 uses the `substring` method that takes one integer argument. The argument specifies the starting index from which characters are copied in the original `String`. The substring returned contains a copy of the characters from the starting index to the end of the `String`. If the index specified as an argument is outside the bounds of the `String`, the program generates a `StringIndexOutOfBoundsException`.

The expression `letters.substring(0, 6)` from line 17 uses the `substring` method that takes two integer arguments. The first argument specifies the starting index from which characters are copied in the original `String`. The second argument specifies the index one beyond the last character to be copied (i.e., copy up to, but not including, that index in the `String`). The substring returned contains copies of the specified characters from the original `String`. If the arguments are outside the bounds of the `String`, the program generates a `StringIndexOutOfBoundsException`.

### 10.9 Concatenating Strings

The `String` method `concat` (Fig. 10.8) concatenates two `String` objects and returns a new `String` object containing the characters from both original `Strings`. If the argument `String` has no characters in it, the original `String` is returned. The expression `s1.concat(s2)` at line 20 appends the characters from the `String s2` to the end of the `String s1`. The original `Strings` to which `s1` and `s2` refer are not modified.

#### Performance Tip 10.2

In programs that frequently perform `String` concatenation, or other `String` modifications, it is more efficient to implement those modifications with class `StringBuffer` (covered in Section 10.13–Section 10.18).

```
1  // Fig. 10.8: StringConcatenation.java
2  // This program demonstrates the String class concat method.
3  // Note that the concat method returns a new String object. It
4  // does not modify the object that invoked the concat method.
5
6  // Java extension packages
7  import javax.swing.*;
8
9  public class StringConcatenation {
10```

Fig. 10.7  `String` class `substring` methods (part 2 of 2).

Fig. 10.8  `String` method `concat` (part 1 of 2).
10.10 Miscellaneous String Methods

Class String provides several methods that return modified copies of Strings or that return a character array. These methods are demonstrated in the application of Fig. 10.9.

Line 22 uses String method replace to return a new String object in which the method replaces every occurrence in String s1 of character 'l' (el) with character 'L'. Method replace leaves the original String unchanged. If there are no occurrences of the first argument in the String, Method replace returns the original String.

Line 26 uses String method toUpperCase to generate a new String object with uppercase letters where corresponding lowercase letters reside in s1. The method returns a new String object containing the converted String and leaves the original String unchanged. If there are no characters to convert to uppercase letters, method toUpperCase returns the original String.

Line 27 uses String method toLowerCase to return a new String object with lowercase letters where corresponding uppercase letters reside in s1. The original String remains unchanged. If there are no characters to convert to lowercase letters, method toLowerCase returns the original String.
// Fig. 10.9: StringMiscellaneous2.java
// This program demonstrates the String methods replace,
// toLowerCase, toUpperCase, trim, toString and toCharArray

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

public class StringMiscellaneous2 {
    // test miscellaneous String methods
    public static void main( String args[] ) {
        String s1 = new String( "hello" ),
            s2 = new String( "GOOD BYE" ),
            s3 = new String( "   spaces   " );

        String output = "s1 = " + s1 + ", s2 = " + s2 + ", s3 = " + s3;

        // test method replace
        output += "\n\nReplace 'l' with 'L' in s1: " +
            s1.replace( 'l', 'L' );

        // test toLowerCase and toUpperCase
        output += "\n\ns1.toUpperCase() = " + s1.toUpperCase() + ", s2.toLowerCase() = " + s2.toLowerCase();

        // test trim method
        output += "\n\ns3 after trim = " + s3.trim() + ";";

        // test toString method
        output += "\n\ns3 = " + s1.toString();

        // test toCharArray method
        char charArray[] = s1.toCharArray();

        output += "\n\ns1 as a character array = ";

        for ( int count = 0; count < charArray.length; ++count )
            output += charArray[ count ];

        JOptionPane.showMessageDialog( null, output,
            "Demonstrating Miscellaneous String Methods",
            JOptionPane.INFORMATION_MESSAGE );

        System.exit( 0 );
    }
} // end class StringMiscellaneous2

Fig. 10.9 Miscellaneous String methods (part 1 of 2).
Line 30 uses `String` method `trim` to generate a new `String` object that removes all white-space characters that appear at the beginning or end of the `String` to which the `trim` message is sent. The method returns a new `String` object containing the `String` without leading or trailing white-space characters. The original `String` remains unchanged.

Line 33 uses `String` method `toString` to return the `String s1`. Why is the `toString` method provided for class `String`? All objects can be converted to `String` in Java by using method `toString`, which originates in class `Object`. If a class that inherits from `Object` (such as `String`) does not override method `toString`, the default version from class `Object` is used. The default version in `Object` creates a `String` consisting of the object’s class name and the hash code for the object. The `toString` method normally is used to express the contents of an object as text. Method `toString` is provided in class `String` to ensure that the proper `String` value is returned.

Line 36 creates a new character array containing a copy of the characters in `String s1` and assigns it to `charArray`.

### 10.11 Using String Method `valueOf`

Class `String` provides a set of `static` class methods that take arguments of various types, convert those arguments to `Strings` and return them as `String` objects. Class `StringValueOf` (Fig. 10.10) demonstrates the `String` class `valueOf` methods.

```java
1  // Fig. 10.10: StringValueOf.java
2  // This program demonstrates the String class valueOf methods.
3
4  // Java extension packages
5  import javax.swing.*;

Fig. 10.10 `String` class `valueOf` methods (part 1 of 2).
```
The expression `String.valueOf( charArray )` from line 23 copies the contents of the character array `charArray` into a new `String` object and returns the new `String`. The expression `String.valueOf( charArray, 3, 3 )` from line 25
copies a portion of the contents of the character array `charArray` into a new `String`. The second argument specifies the starting index from which the characters are copied. The third argument specifies the number of characters to copy.

There are seven other versions of method `valueOf`, which take arguments of type `boolean`, `char`, `int`, `long`, `float`, `double` and `Object`, respectively. These are demonstrated in lines 26–32 of the program. Note that the version of `valueOf` that takes an `Object` as an argument can do so because all `Object`s can be converted to `Strings` with method `toString`.

10.12 String Method `intern`

Comparing large `String` objects is a relatively slow operation. `String` method `intern` can improve `String` comparison performance. When `String` method `intern` is invoked on a `String` object, it returns a reference to a `String` object that is guaranteed to have the same contents as the original `String`. Class `String` maintains the resulting `String` during program execution. Subsequently, if the program invokes method `intern` on other `String` objects with contents identical to the original `String`, method `intern` returns a reference to the copy of the `String` maintained in memory by class `String`. If a program uses `intern` on extremely large `Strings`, the program can compare those `Strings` faster by using the `==` operator, which simply compares two references—a fast operation. Using standard `String` methods such as `equals` and `equalsIgnoreCase` can be slower, because they compare corresponding characters in each `String`. For large `Strings`, this is a time-consuming, iterative operation. The program of Fig. 10.11 demonstrates the `intern` method.

The program declares five `String` references—`s1`, `s2`, `s3`, `s4` and `output`. `Strings s1` and `s2` are initialized with new `String` objects that each contain a copy of "hello". The first `if` structure (lines 20–23) uses operator `==` to determine that `Strings s1` and `s2` are the same object. References `s1` and `s2` refer to different objects, because they were initialized with new `String` objects.

The second `if` structure (lines 26–29) uses method `equals` to determine that the contents of `Strings s1` and `s2` are equal. They were each initialized with copies of "hello", so they have the same contents.

```java
// Fig. 10.11: StringIntern.java
// This program demonstrates the intern method
// of the String class.

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

public class StringIntern {
  // test String method intern
  public static void main( String args[] )
  {
    String s1, s2, s3, s4, output;
```
s1 = new String( "hello" );
s2 = new String( "hello" );

// test strings to determine if they are same
// String object in memory
if ( s1 == s2 )
    output = "s1 and s2 are the same object in memory";
else
    output = "s1 and s2 are not the same object in memory";

// test strings for equality of contents
if ( s1.equals( s2 ) )
    output += "\ns1 and s2 are equal";
else
    output += "\ns1 and s2 are not equal";

// use String intern method to get a unique copy of "hello" referred to by both s3 and s4
s3 = s1.intern();
s4 = s2.intern();

// test strings to determine if they are same
// String object in memory
if ( s3 == s4 )
    output += "\ns3 and s4 are the same object in memory";
else
    output += "\ns3 and s4 are not the same object in memory";

// determine if s1 and s3 refer to same object
if ( s1 == s3 )
    output += "\ns1 and s3 are the same object in memory";
else
    output += "\ns1 and s3 are not the same object in memory";

// determine if s2 and s4 refer to same object
if ( s2 == s4 )
    output += "\ns2 and s4 are the same object in memory";
else
    output += "\ns2 and s4 are not the same object in memory";

// determine if s1 and s4 refer to same object
if ( s1 == s4 )
    output += "\ns1 and s4 are the same object in memory";
else
    output += "\ns1 and s4 are not the same object in memory";
Line 33 uses method \texttt{intern} to get a reference to a \texttt{String} with the same contents as object \texttt{s1} and assigns the reference to \texttt{s3}. The \texttt{String} to which \texttt{s3} refers is maintained in memory by class \texttt{String}. Line 34 also uses method \texttt{intern} to get a reference to a \texttt{String} object. However, because \texttt{String s1} and \texttt{String s2} have the same contents, the reference returned by this call to \texttt{intern} is a reference to the same \texttt{String} object returned by \texttt{s1.intern()}. The third \texttt{if} structure (lines 38–42) uses operator \texttt{==} to determine that \texttt{Strings s3} and \texttt{s4} refer to the same object.

The fourth \texttt{if} structure (lines 45–50) uses operator \texttt{==} to determine that \texttt{Strings s1} and \texttt{s3} are \textit{not} the same object. Technically, they could refer to the same object, but they are not guaranteed to refer to the same object unless the objects they refer to were returned by calls to \texttt{intern} on \texttt{Strings} with the same contents. In this case, \texttt{s1} refers to the \texttt{String} it was assigned in method \texttt{main} and \texttt{s3} refers to the \texttt{String} with the same contents maintained by class \texttt{String}.

The fifth \texttt{if} structure (lines 53–57) uses operator \texttt{==} to determine that \texttt{Strings s2} and \texttt{s4} are \textit{not} the same object, because the second \texttt{intern} call results in a reference to the same object returned by \texttt{s1.intern()}, not \texttt{s2}. Similarly, the sixth \texttt{if} structure (lines 60–64) uses operator \texttt{==} to determine that \texttt{Strings s1} and \texttt{s4} are not the same object, because the second \texttt{intern} call results in a reference to the same object returned by \texttt{s1.intern()}, not \texttt{s1}.

10.13 StringBuffer Class

The \texttt{String} class provides many capabilities for processing \texttt{Strings}. However, once a \texttt{String} object is created, its contents can never change. The next several sections discuss the features of class \texttt{StringBuffer} for creating and manipulating dynamic string information—i.e., modifiable \texttt{Strings}. Every \texttt{StringBuffer} is capable of storing a number of characters specified by its capacity. If the capacity of a \texttt{StringBuffer} is exceeded, the capacity is automatically expanded to accommodate the additional charac-
ters. As we will see, class `StringBuffer` is also used to implement operators `+` and `+=` for `String` concatenation.

**Performance Tip 10.3**

`String` objects are constant strings and `StringBuffer` objects are modifiable strings. Java distinguishes constant strings from modifiable strings for optimization purposes; in particular, Java can perform certain optimizations involving `String` objects (such as sharing one `String` object among multiple references) because it knows these objects will not change.

**Performance Tip 10.4**

When given the choice between using a `String` object to represent a string versus a `StringBuffer` object to represent that string, always use a `String` object if indeed the object will not change; this improves performance.

**Common Programming Error 10.4**

Invoking `StringBuffer` methods that are not methods of class `String` on `String` objects is a syntax error.

### 10.14 StringBuffer Constructors

Class `StringBuffer` provides three constructors (demonstrated in Fig. 10.12). Line 14 uses the default `StringBuffer` constructor to create a `StringBuffer` with no characters in it and an initial capacity of 16 characters. Line 15 uses `StringBuffer` constructor that takes an integer argument to create a `StringBuffer` with no characters in it and the initial capacity specified in the integer argument (i.e., 10). Line 16 uses the `StringBuffer` constructor that takes a `String` argument to create a `StringBuffer` containing the characters of the `String` argument. The initial capacity is the number of characters in the `String` argument plus 16.

The statement on lines 18–21 uses `StringBuffer` method `toString` to convert the `StringBuffers` into `String` objects that can be displayed with `drawString`. Note the use of operator `+` to concatenate `Strings` for output. In Section 10.17, we discuss how Java uses `StringBuffers` to implement the `+` and `+=` operators for `String` concatenation.

```java
public class StringBufferConstructors {
    public static void main( String args[] ) {
        StringBuffer buffer1, buffer2, buffer3;
```

---

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
14 buffer1 = new StringBuffer();
15 buffer2 = new StringBuffer( 10 );
16 buffer3 = new StringBuffer( "hello" );
17
18 String output =
19 "buffer1 = " + buffer1.toString() + "\n" +
20 "buffer2 = " + buffer2.toString() + "\n" +
21 "buffer3 = " + buffer3.toString() + "\n";
22
23 JOptionPane.showMessageDialog( null, output,
24 "Demonstrating StringBuffer Class Constructors",
25 JOptionPane.INFORMATION_MESSAGE );
26
27 System.exit( 0 );
28 }
29 }
30 } // end class StringBufferConstructors

Fig. 10.12 StringBuffer class constructors (part 2 of 2).

10.15 StringBuffer Methods length, capacity, setLength and ensureCapacity

Class StringBuffer provides the length and capacity methods to return the number of characters currently in a StringBuffer and the number of characters that can be stored in a StringBuffer without allocating more memory, respectively. Method ensureCapacity is provided to allow the programmer to guarantee that a StringBuffer has a minimum capacity. Method setLength is provided to enable the programmer to increase or decrease the length of a StringBuffer. The program of Fig. 10.13 demonstrates these methods.

// Fig. 10.13: StringBufferCapLen.java
// This program demonstrates the length and
// capacity methods of the StringBuffer class.

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

public class StringBufferCapLen {

   // test StringBuffer methods for capacity and length
   public static void main( String args[] )
   {

Fig. 10.13 StringBuffer length and capacity methods (part 1 of 2).
The program contains one `StringBuffer` called `buffer`. Lines 13–14 of the program use the `StringBuffer` constructor that takes a `String` argument to instantiate and initialize the `StringBuffer` with "Hello, how are you?". Lines 16–18 append to `output` the contents, the length and the capacity of the `StringBuffer`. Notice in the output window that the capacity of the `StringBuffer` is initially 35. Remember that the `StringBuffer` constructor that takes a `String` argument creates a `StringBuffer` object with an initial capacity that is the length of the `String` passed as an argument plus 16.

Line 20 expands the capacity of the `StringBuffer` to a minimum of 75 characters. Actually, if the original capacity is less than the argument, the method ensures a capacity that is the greater of the number specified as an argument or twice the original capacity plus 2. If the `StringBuffer`’s current capacity is more than the specified capacity, the `StringBuffer`’s capacity remains unchanged.

Line 23 uses method `setLength` to set the length of the `StringBuffer` to 10. If the specified length is less than the current number of characters in the `StringBuffer`,
the characters are truncated to the specified length (i.e., the characters in the `StringBuffer` after the specified length are discarded). If the specified length is greater than the number of characters currently in the `StringBuffer`, null characters (characters with the numeric representation 0) are appended to the `StringBuffer` until the total number of characters in the `StringBuffer` is equal to the specified length.

### 10.16 StringBuffer Methods `charAt`, `setCharAt`, `getChars` and `reverse`

Class `StringBuffer` provides the `charAt`, `setCharAt`, `getChars` and `reverse` methods to manipulate the characters in a `StringBuffer`. Method `charAt` takes an integer argument and returns the character in the `StringBuffer` at that index. Method `setCharAt` takes an integer and a character argument and sets the character at the specified position to the character argument. The index specified in the `charAt` and `setCharAt` methods must be greater than or equal to 0 and less than the `StringBuffer` length; otherwise, a `StringIndexOutOfBoundsException` is generated.

#### Common Programming Error 10.5

Attempting to access a character that is outside the bounds of a `StringBuffer` (i.e., an index less than 0 or an index greater than or equal to the `StringBuffer`'s length) results in a `StringIndexOutOfBoundsException`.

Method `getChars` returns a character array containing a copy of the characters in the `StringBuffer`. This method takes four arguments—the starting index from which characters should be copied in the `StringBuffer`, the index one past the last character to be copied from the `StringBuffer`, the character array into which the characters are to be copied and the starting location in the character array where the first character should be placed. Method `reverse` reverses the contents of the `StringBuffer`. Each of these methods is demonstrated in Fig. 10.14.

```java
// Fig. 10.14: StringBufferChars.java
// The charAt, setCharAt, getChars, and reverse methods
// of class StringBuffer.

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

public class StringBufferChars {

    // test StringBuffer character methods
    public static void main( String args[] )
    {
        StringBuffer buffer = new StringBuffer( "hello there" );
        String output = "buffer = " + buffer.toString() +
                        "\nCharacter at 0: " + buffer.charAt( 0 ) +
                        "\nCharacter at 4: " + buffer.charAt( 4 );

    }
}
```

**Fig. 10.14** `StringBuffer` class character manipulation methods.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
**10.17 StringBuffer append Methods**

Class `StringBuffer` provides 10 overloaded `append` methods to allow various data-type values to be added to the end of a `StringBuffer`. Versions are provided for each of the primitive data types and for character arrays, `Strings` and `Objects`. (Remember that method `toString` produces a `String` representation of any `Object`.) Each of the methods takes its argument, converts it to a `String` and appends it to the `StringBuffer`. The `append` methods are demonstrated in Fig. 10.15. [Note: Line 20 specifies the literal value `2.5f` as the initial value of a `float` variable. Normally, Java treats a floating-point literal value as type `double`. Appending the letter `f` to the literal `.5` indicates to the compiler that `2.5` should be treated as type `float`. Without this indication the compiler generates a syntax error, because a `double` value cannot be assigned directly to a `float` variable in Java.]
// Fig. 10.15: StringBufferAppend.java
// This program demonstrates the append
// methods of the StringBuffer class.

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

public class StringBufferAppend {
    // test StringBuffer append methods
    public static void main(String args[]) {
        Object o = "hello";
        String s = "good bye";
        char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
        boolean b = true;
        char c = 'Z';
        int i = 7;
        long l = 10000000;
        float f = 2.5f;
        double d = 33.333;
        StringBuffer buffer = new StringBuffer();

        buffer.append(o);
        buffer.append(" ");

        buffer.append(s);
        buffer.append(" ");

        buffer.append(charArray);
        buffer.append(" ");

        buffer.append(charArray, 0, 3);
        buffer.append(" ");

        buffer.append(b);
        buffer.append(" ");

        buffer.append(c);
        buffer.append(" ");

        buffer.append(i);
        buffer.append(" ");

        buffer.append(l);
        buffer.append(" ");

        buffer.append(f);
        buffer.append(" ");

        buffer.append(d);

        JOptionPane.showMessageDialog(null,
            "buffer = " + buffer.toString(),
            "Demonstrating StringBuffer append Methods",
            JOptionPane.INFORMATION_MESSAGE);

        System.exit(0);
    }
}

// end StringBufferAppend

Fig. 10.15 StringBuffer class append methods (part 1 of 2).
Actually, `StringBuffer` and the `append` methods are used by the compiler to implement the `+` and `+=` operators for concatenating `String`s. For example, assuming the following declarations:

```java
String string1 = "hello";
String string2 = "BC"
int value = 22;
```

the statement

```java
String s = string1 + string2 + value;
```

concatenates "hello", "BC" and 22. The concatenation is performed as follows:

```java
new StringBuffer().append( "hello" ).append( "BC" ).append( 22 ).toString();
```

First, Java creates a `StringBuffer`, then appends to the `StringBuffer` the `String"hello", the `String"BC"` and the integer 22. Next, `StringBuffer`'s `toString` converts the `StringBuffer` to a `String` representation and the result is assigned to `String s`. The statement

```java
s += "!";
```

is performed as follows:

```java
s = new StringBuffer().append( s ).append( "!" ).toString();
```

First, Java creates a `StringBuffer`, then appends to the `StringBuffer` the current contents of `s` followed by "!". Next, `StringBuffer`'s `toString` converts the `StringBuffer` to a `String` representation and the result is assigned to `s`

### 10.18 StringBuffer Insertion and Deletion Methods

Class `StringBuffer` provides nine overloaded `insert` methods to allow various data-type values to be inserted at any position in a `StringBuffer`. Versions are provided for each of the primitive data types and for character arrays, `Strings` and `Objects`. (Remember that method `toString` produces a `String` representation of any `Object`.) Each of the methods takes its second argument, converts it to a `String` and inserts it preceding the index specified by the first argument. The index specified by the first argument must be greater than or equal to 0 and less than the length of the `StringBuffer`; otherwise, a `StringIndexOutOfBoundsException` is generated. Class `StringBuffer` also provides methods `delete` and `deleteCharAt` for deleting characters at
any position in a `StringBuffer`. Method `delete` takes two arguments—the starting subscript and the subscript one past the end of the characters to delete. All characters beginning at the starting subscript up to, but not including the ending subscript are deleted. Method `deleteCharAt` takes one argument—the subscript of the character to delete. Invalid subscripts cause both methods to throw a `StringIndexOutOfBoundsException`. The `insert` and delete methods are demonstrated in Fig. 10.16.

```java
// Fig. 10.16: StringBufferInsert.java
// This program demonstrates the insert and delete methods of class StringBuffer.

import javax.swing.*;

public class StringBufferInsert {

    public static void main( String args[] )
    {
        Object o = "hello";
        String s = "good bye";
        char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
        boolean b = true;
        char c = 'K';
        int i = 7;
        long l = 1000000;
        float f = 2.5f;
        double d = 33.333;
        StringBuffer buffer = new StringBuffer();

        buffer.insert( 0, o );
        buffer.insert( 0, " " );
        buffer.insert( 0, s );
        buffer.insert( 0, " " );
        buffer.insert( 0, charArray );
        buffer.insert( 0, " " );
        buffer.insert( 0, b );
        buffer.insert( 0, " " );
        buffer.insert( 0, c );
        buffer.insert( 0, " " );
        buffer.insert( 0, i );
        buffer.insert( 0, " " );
        buffer.insert( 0, l );
        buffer.insert( 0, " " );
        buffer.insert( 0, f );
        buffer.insert( 0, " " );
        buffer.insert( 0, d );

        String output =
            "buffer after inserts:\n" + buffer.toString();
    }
}
```

Fig. 10.16 `StringBuffer` class `insert` and `delete` methods (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
10.19 Character Class Examples

Java provides a number of classes that enable primitive variables to be treated as objects. The classes are Boolean, Character, Double, Float, Byte, Short, Integer and Long. These classes (except Boolean and Character) are derived from Number. These eight classes are known as type wrappers, and they are part of the java.lang package. Objects of these classes can be used anywhere in a program that an Object or a Number is expected. In this section, we present class Character—the type-wrapper class for characters.

Most Character class methods are static, take at least a character argument and perform either a test or a manipulation of the character. This class also contains a constructor that receives a char argument to initialize a Character object and several non-static methods. Most of the methods of class Character are presented in the next three examples. For more information on class Character (and all the wrapper classes), see the java.lang package in the Java API documentation.

Figure 10.17 demonstrates some static methods that test characters to determine whether they are a specific character type and the static methods that perform case conversions on characters. Each method is used in method buildOutput of class StaticCharMethods. You can enter any character and apply the preceding methods to the character. Note the use of inner classes for the event handling as demonstrated in Chapter 9.
Line 63 uses `Character` method `isDefined` to determine if character `c` is defined in the Unicode character set. If so, the method returns `true`; otherwise, it returns `false`.

Fig. 10.17 static character testing methods and case conversion methods of class `Character` (part 1 of 3).
container.add( inputField );

outputArea = new JTextArea( 10, 20 );
container.add( outputArea );

setSize( 300, 250 ); // set the window size
show(); // show the window

// display character info in outputArea
public void buildOutput()
{
    outputArea.setText(
        "is defined: " + Character.isDefined( c ) + 
        "\nis digit: " + Character.isDigit( c ) + 
        "\nis Java letter: " + 
        Character.isJavaIdentifierStart( c ) + 
        "\nis Java letter or digit: " + 
        Character.isJavaIdentifierPart( c ) + 
        "\nis letter: " + Character.isLetter( c ) + 
        "\nis letter or digit: " + 
        Character.isLetterOrDigit( c ) + 
        "\nis lower case: " + Character.isLowerCase( c ) + 
        "\nis upper case: " + Character.isUpperCase( c ) + 
        "\nto upper case: " + Character.toUpperCase( c ) + 
        "\nto lower case: " + Character.toLowerCase( c ) );
}

// execute application
public static void main( String args[] )
{
    StaticCharMethods application = new StaticCharMethods();
    application.addWindowListener(
        new WindowAdapter()
        {
        // handle event when user closes window
            public void windowClosing( WindowEvent windowEvent )
            {
                System.exit( 0 );
            }
        } // end anonymous inner class
    ); // end call to addWindowListener
}
} // end class StaticCharMethods

Fig. 10.17 static character testing methods and case conversion methods of class Character (part 2 of 3).
Line 64 uses `Character` method `isDigit` to determine whether character `c` is a defined Unicode digit. If so, the method returns `true`; otherwise, it returns `false`.

Line 66 uses `Character` method `isJavaIdentifierStart` to determine whether `c` is a character that can be used as the first character of an identifier in Java—i.e., a letter, an underscore (_) or a dollar sign ($). If so, the method returns `true`; otherwise, it returns `false`. Line 68 uses method `Character` method `isJavaIdentifierPart` to determine whether character `c` is a character that can be used in an identifier in Java—i.e., a digit, a letter, an underscore (_) or a dollar sign ($). If so, the method returns `true`; otherwise, it returns `false`.

Line 69 uses method `Character` method `isLetter` to determine whether character `c` is a letter. If so, the method returns `true`; otherwise, it returns `false`. Line 71 uses method `Character` method `isLetterOrDigit` to determine whether character `c` is a letter or a digit. If so, the method returns `true`; otherwise, it returns `false`.

Line 72 uses method `Character` method `isLowerCase` to determine whether character `c` is a lowercase letter. If so, the method returns `true`; otherwise, it returns `false`. Line 73 uses method `Character` method `isUpperCase` to determine whether character `c` is an uppercase letter. If so, the method returns `true`; otherwise, it returns `false`.

Line 74 uses method `Character` method `toUpperCase` to convert the character `c` to its uppercase equivalent. The method returns the converted character if the character has an uppercase equivalent; otherwise, the method returns its original argument. Line 75 uses method `Character` method `toLowerCase` to convert the character `c` to its lowercase equivalent.
equivalent. The method returns the converted character if the character has a lowercase equivalent; otherwise, the method returns its original argument.

Figure 10.18 demonstrates static Character methods digit and forDigit, which perform conversions between characters and digits in different number systems. Common number systems include decimal (base 10), octal (base 8), hexadecimal (base 16) and binary (base 2). The base of a number is also known as its radix. For more information on conversions between number systems, see Appendix E.

Line 52 uses method forDigit to convert the integer digit into a character in the number system specified by the integer radix (also known as the base of the number). For example, the integer 13 in base 16 (the radix) has the character value 'd'. Note that the lowercase and uppercase letters are equivalent in number systems.

Line 77 uses method digit to convert the character c into an integer in the number system specified by the integer radix (i.e., the base of the number). For example, the character 'A' in base 16 (the radix) has the integer value 10.

```java
// Fig. 10.18: StaticCharMethods2.java
// Demonstrates the static character conversion methods
// of class Character from the java.lang package.

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

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

public class StaticCharMethods2 extends JFrame {
    private char c;
    private int digit, radix;
    private JLabel prompt1, prompt2;
    private JTextField input, radixField;
    private JButton toChar, toInt;

    public StaticCharMethods2()
    {
        super( "Character Conversion Methods" );
        // set up GUI and event handling
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        prompt1 = new JLabel( "Enter a digit or character " );
        input = new JTextField( 5 );
        container.add( prompt1 );
        container.add( input );

        prompt2 = new JLabel( "Enter a radix " );
        radixField = new JTextField( 5 );
        container.add( prompt2 );
        container.add( radixField );
```
toChar = new JButton( "Convert digit to character" );

toChar.addActionListener(

   // anonymous inner class
   new ActionListener() {

     // handle toChar JButton event
     public void actionPerformed( ActionEvent actionEvent )
     {
       digit = Integer.parseInt( input.getText() );
       radix =
           Integer.parseInt( radixField.getText() );
       JOptionPane.showMessageDialog( null,
           "Convert digit to character: " +
           Character.forDigit( digit, radix ) );
     }
   }  // end anonymous inner class
); // end call to addActionListener

container.add( toChar );

toInt = new JButton( "Convert character to digit" );

toInt.addActionListener(

   // anonymous inner class
   new ActionListener() {

     // handle toInt JButton event
     public void actionPerformed( ActionEvent actionEvent )
     {
       String s = input.getText();
       c = s.charAt( 0 );
       radix =
           Integer.parseInt( radixField.getText() );
       JOptionPane.showMessageDialog( null,
           "Convert character to digit: " +
           Character.digit( c, radix ) );
     }
   }  // end anonymous inner class
); // end call to addActionListener

container.add( toInt );

setSize( 275, 150 );  // set the window size
show();               // show the window

Fig. 10.18 Character class static conversion methods (part 2 of 3).
The program in Fig. 10.19 demonstrates the nonstatic methods of class Character—the constructor, charValue, toString, hashCode and equals.

Fig. 10.19 Non-static methods of class Character (part 1 of 2).

```java
// Fig. 10.19: OtherCharMethods.java
// Demonstrate the non-static methods of class Character from the java.lang package.
// Java extension packages
import javax.swing.*;
```
Lines 15–16 instantiate two `Character` objects and pass character literals to the constructor to initialize those objects.

Line 18 uses `Character` method `charValue` to return the `char` value stored in `Character` object `c1`. Line 19 returns a `String` representation of `Character` object `c2` using method `toString`.

Lines 20–21 perform `hashCode` calculations on the `Character` objects `c1` and `c2`, respectively. Remember that hash code values are used to store objects in hash tables for fast lookup capabilities (see Chapter 20).
The condition in the `if` structure on line 23 uses method `equals` to determine whether the object `c1` has the same contents as the object `c2` (i.e., the characters inside each object are equal).

### 10.20 Class `StringTokenizer`

When you read a sentence, your mind breaks the sentence into individual words and punctuation, or *tokens*, each of which conveys meaning to you. Compilers also perform tokenization. They break up statements into individual pieces like keywords, identifiers, operators and other elements of a programming language. In this section, we study Java’s `StringTokenizer` class (from package `java.util`), which breaks a string into its component tokens. Tokens are separated from one another by delimiters, typically whitespace characters such as blank, tab, newline and carriage return. Other characters can also be used as delimiters to separate tokens. The program in Fig. 10.20 demonstrates class `StringTokenizer`. The window for class `TokenTest` displays a `JTextField` where the user types a sentence to tokenize. Output in this program is displayed in a `JTextArea`.

When the user presses the `Enter` key in the `JTextField`, method `actionPerformed` (lines 37–49) is invoked. Lines 39–40 assign `String` reference `stringToTokenize` the value in the text in the `JTextField` returned by calling `event.getActionCommand()`. Next, lines 41–42 create an instance of class `StringTokenizer`. This `StringTokenizer` constructor takes a `String` argument and creates a `StringTokenizer` for `stringToTokenize` that will use the default delimiter string " 
\t\r" consisting of a space, a newline, a tab and a carriage return for tokenization. There are two other constructors for class `StringTokenizer`. In the version that takes two `String` arguments, the second `String` is the delimiter `String`. In the version that takes three arguments, the second `String` is the delimiter `String` and the third argument (a `boolean`) determines whether the delimiters are also returned as tokens (only if the argument is `true`). This is useful if you need to know what the delimiters are.

```java
// Fig. 10.20: TokenTest.java
// Testing the StringTokenizer class of the java.util package

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

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

public class TokenTest extends JFrame {
    private JLabel promptLabel;
    private JTextField inputField;
    private JTextArea outputArea;
}
```

---

Fig. 10.20 Tokenizing strings with a `StringTokenizer` object (part 1 of 3).
Chapter 10  Strings and Characters  577

Fig. 10.20 Tokenizing strings with a StringTokenizer object (part 2 of 3).

// set up GUI and event handling
public TokenTest() {
    super( "Testing Class StringTokenizer" );

    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    promptLabel =
        new JLabel( "Enter a sentence and press Enter" );
    container.add( promptLabel );

    inputField = new JTextField( 20 );

    inputField.addActionListener(
        new ActionListener() {
            // handle text field event
            public void actionPerformed( ActionEvent event ) {
                String stringToTokenize =
                    event.getActionCommand();
                StringTokenizer tokens =
                    new StringTokenizer( stringToTokenize );

                outputArea.setText( "Number of elements: " +
                    tokens.countTokens() + "\nThe tokens are:\n" );

                while ( tokens.hasMoreTokens() )
                    outputArea.append( tokens.nextToken() + "\n" );
            }
        } // end anonymous inner class
    ); // end call to addActionListener

    container.add( inputField );

    outputArea = new JTextArea( 10, 20 );
    outputArea.setEditable( false );
    container.add( new JScrollPane( outputArea ) );

    setSize( 275, 260 ); // set the window size
    show();               // show the window
}

// execute application
public static void main( String args[] ) {
    TokenTest application = new TokenTest();
application.addWindowListener(
    // anonymous inner class
    new WindowAdapter() {
        // handle event when user closes window
        public void windowClosing( WindowEvent windowEvent )
        {
            System.exit( 0 );
        }
    } // end anonymous inner class
); // end call to addWindowListener
} // end method main
} // end class TokenTest

The statement at lines 44–45 uses the StringTokenizer method countTokens to determine the number of tokens in the String to be tokenized.

The condition in the while structure at lines 47–48 StringTokenizer method hasMoreTokens to determine whether there are more tokens in the String being tokenized. If so, the append method is invoked for the JTextArea outputArea to append the next token to the String in the JTextArea. The next token is obtained with a call to StringTokenizer method nextToken that returns a String. The token is output followed by a newline character, so subsequent tokens appear on separate lines.

If you would like to change the delimiter String while tokenizing a String, you may do so by specifying a new delimiter string in a nextToken call as follows:

tokens.nextToken( newDelimiterString );

This feature is not demonstrated in the program.
10.21 Card Shuffling and Dealing Simulation

In this section, we use random number generation to develop a card shuffling and dealing simulation program. This program can then be used to implement programs that play specific card games.

We develop application `DeckOfCards` (Fig. 10.21), which creates a deck of 52 playing cards using `Card` objects, then enables the user to deal each card by clicking on a “Deal card” button. Each card dealt is displayed in a `JTextField`. The user can also shuffle the deck at any time by clicking on a “Shuffle cards” button.

```java
public class DeckOfCards extends JFrame {
    private Card deck[];
    private int currentCard;
    private JButton dealButton, shuffleButton;
    private JTextField displayField;
    private JLabel statusLabel;

    public DeckOfCards() {
        super( "Card Dealing Program" );

        String suits[] = {
            "Hearts", "Diamonds", "Clubs", "Spades" }

        deck = new Card[ 52 ];
        currentCard = -1;

        // populate deck with Card objects
        for ( int count = 0; count < deck.length; count++ )
            deck[ count ] = new Card( faces[ count % 13 ],
                        suits[ count / 13 ] );

        // set up GUI and event handling
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );
        dealButton = new JButton( "Deal card" );
```

Fig. 10.21 Card dealing program (part 1 of 4).
dealButton.addActionListener(
    // anonymous inner class
    new ActionListener() {
        // deal one card
        public void actionPerformed( ActionEvent actionEvent ) {
            Card dealt = dealCard();
            if ( dealt != null ) {
                displayField.setText( dealt.toString() );
                statusLabel.setText( "Card #: " + currentCard );
            } else {
                displayField.setText( "NO MORE CARDS TO DEAL" );
                statusLabel.setText( "Shuffle cards to continue" );
            }
        }
    }  // end anonymous inner class
    ); // end call to addActionListener
    container.add( dealButton );

    shuffleButton = new JButton( "Shuffle cards" );
    shuffleButton.addActionListener(
        // anonymous inner class
        new ActionListener() {
            // shuffle deck
            public void actionPerformed( ActionEvent actionEvent ) {
                displayField.setText( "SHUFFLING ..." );
                shuffle();
                displayField.setText( "DECK IS SHUFFLED" );
            }
        }  // end anonymous inner class
    ); // end call to addActionListener
    container.add( shuffleButton );

displayField = new JTextField( 20 );
displayField.setEditable( false );
container.add( displayField );

statusLabel = new JLabel();
Chapter 10

Strings and Characters

581

container.add( statusLabel );

setSize( 275, 120 ); // set window size
show(); // show window

} // shuffle deck of cards with one-pass algorithm

public void shuffle()
{
    currentCard = -1;

    // for each card, pick another random card and swap them
    for ( int first = 0; first < deck.length; first++ ) {
        int second = ( int ) ( Math.random() * 52 );
        Card temp = deck[ first ];
        deck[ first ] = deck[ second ];
        deck[ second ] = temp;
    }

    dealButton.setEnabled( true );
}

// deal one card

public Card dealCard()
{
    if ( ++currentCard < deck.length )
        return deck[ currentCard ];
    else {
        dealButton.setEnabled( false );
        return null;
    }
}

// execute application

public static void main( String args[] )
{
    DeckOfCards app = new DeckOfCards();
    app.addWindowListener( new WindowAdapter() {
        // anonymous inner class
        public void windowClosing( WindowEvent windowEvent )
        {
            System.exit( 0 );
        }
    } // end anonymous inner class
    ); // end call to addWindowListener

Fig. 10.21 Card dealing program (part 3 of 4).
Class `Card` (lines 153–170) contains two `String` instance variables—`face` and `suit`—that are used to store references to the face name and suit name for a specific `Card`. The constructor for the class receives two `Strings` that it uses to initialize `face` and `suit`. Method `toString` is provided to create a `String` consisting of the `face` of the card, the string " of " and the `suit` of the card.

Class `DeckOfCards` (lines 11–150) consists of an array `deck` of 52 `Cards`, an integer `currentCard` representing the most recently dealt card in the deck array (–1 if no cards have been dealt yet) and the GUI components used to manipulate the deck of cards. The constructor method of the application instantiates the `deck` array (line 29) and uses the `for` structure at lines 33–35 to fill the `deck` array with `Cards`. Note that each `Card` is instantiated and initialized with two `Strings`—one from the `faces` array (`Strings: "Ace" through "King"`) and one from the `suits` array ("Hearts", "Diamonds", "Clubs" and "Spades"). The calculation `count % 13` always results in a value from 0 to 12 (the
thirteen subscripts of the faces array), and the calculation count / 13 always results in a value from 0 to 3 (the four subscripts in the suits array). When the deck array is initialized, it contains the cards with faces ace through king in order for each suit.

When the user clicks the Deal card button, method actionPerformed at lines 48–62 invokes method dealCard (defined at lines 118–126) to get the next card in the array. If the deck is not empty, a Card object reference is returned; otherwise, null is returned. If the reference is not null, lines 53–54 display the Card in the JTextField displayField and display the card number in the JLabel statusLabel. If the reference returned by dealCard was null, the String “NO MORE CARDS TO DEAL!” is displayed in the JTextField and the String “Shuffle cards to continue” is displayed in the JLabel.

When the user clicks the Shuffle cards button, its actionPerformed method at lines 77–82 invokes method shuffle (defined on lines 102–115) to shuffle the cards. The method loops through all 52 cards (array subscripts 0 to 51). For each card, a number between 0 and 51 is picked randomly. Next, the current Card object and the randomly selected Card object are swapped in the array. A total of only 52 swaps are made in a single pass of the entire array, and the array of Card objects is shuffled! When the shuffling is complete, the String “DECK IS SHUFFLED” is displayed in the JTextField.

Notice the use of method setEnabled at lines 114 and 123 to activate and deactivate the dealButton. Method setEnabled can be used on many GUI components. When it is called with a false argument, the GUI component for which it is called is disabled so the user cannot interact with it. To reactivate the button, method setEnabled is called with a true argument.

10.22 (Optional Case Study) Thinking About Objects: Event Handling

Objects do not ordinarily perform their operations spontaneously. Rather, a specific operation is normally invoked when a sending object (a client object) sends a message to a receiving object (a server object) requesting that the receiving object perform that specific operation. In earlier sections, we mentioned that objects interact by sending and receiving messages. We began to model the behavior of our elevator system by using statechart and activity diagrams in Section 5.11 and collaboration diagrams in Section 7.10. In this section, we discuss how the objects of the elevator system interact.

Events

In Fig. 7.24, we presented an example of a person pressing a button by sending a press-Button message to the button—specifically, the Person object called method press-Button of the Button object. This message describes an action that is currently happening; in other words, the Person presses a Button. In general, the message name structure is a verb preceding a noun—e.g., the name of the pressButton message consists of the verb “press” followed by the noun “button.”

An event is a message that notifies an object of an action that has already happened. For example, in this section, we modify our simulation so the Elevator sends an elevatorArrived event to the Elevator’s Door when the Elevator arrives at a Floor. In Section 7.10, the Elevator opens this Door directly by sending an openDoor message. Listening for an elevatorArrived event allows the Door to determine the actions
to take when the Elevator has arrived, such as notifying the Person that the Door has opened. This reinforces the OOD principle of encapsulation and models the real world more closely. In reality, the door—not the elevator—“notifies” a person of a door’s opening.

Notice that the event-naming structure is the inverse of the first type of message’s naming structure. By convention, the event name consists of the noun preceding the verb. For instance, the elevatorArrived event name consists of the noun “elevator” preceding the verb “arrived.”

In our simulation, we create a superclass called ElevatorModelEvent (Fig. 10.23) that represents an event in our model. ElevatorModelEvent contains a Location reference (line 11) that represents the location where the event was generated and an Object reference (line 14) to the source of the event. In our simulation, objects use instances of ElevatorModelEvent to send events to other objects. When an object receives an event, that object may use method getLocation (lines 31–34) and method getSource (lines 43–46) to determine the event’s location and origin.

```
// ElevatorModelEvent.java
// Basic event packet holding Location object
package com.deitel.jhtp4.elevator.event;

// Deitel packages
import com.deitel.jhtp4.elevator.model.*;

public class ElevatorModelEvent {

    // Location that generated ElevatorModelEvent
    private Location location;

    // source of generated ElevatorModelEvent
    private Object source;

    // ElevatorModelEvent constructor sets Location
    public ElevatorModelEvent( Object source,
        Location location )
    {
        setSource( source );
        setLocation( location );
    }

    // set ElevatorModelEvent Location
    public void setLocation( Location eventLocation )
    {
        location = eventLocation;
    }

    // get ElevatorModelEvent Location
    public Location getLocation()
    {
        return location;
    }
}
```

Fig. 10.23 Class ElevatorModelEvent is the superclass for all other event classes in our model (part 1 of 2).
For example, a Door may send an ElevatorModelEvent to a Person when opening or closing, and the Elevator may send an ElevatorModelEvent informing a person of a departure or arrival. Having different objects send the same event type to describe different actions could be confusing. To eliminate ambiguity as we discuss what events are sent by objects, we create several ElevatorModelEvent subclasses in Fig. 10.24, so we will have an easier time associating each event with its sender. According to Fig. 10.24, classes BellEvent, PersonMoveEvent, LightEvent, ButtonEvent, ElevatorMoveEvent and DoorEvent are subclasses of class ElevatorModelEvent. Using these event subclasses, a Door sends a different event (a DoorEvent) than does a Button (which sends a ButtonEvent). Figure 10.25 displays the triggering actions of the subclass events. Note that all actions in Fig. 10.25 appear in the form “noun” + “verb”.

```java
35    // set ElevatorModelEvent source
36    private void setSource( Object eventSource )
37    {
38        source = eventSource;
39    }
40
41    // get ElevatorModelEvent source
42    public Object getSource()
43    {
44        return source;
45    }
46    }
```

Fig. 10.23 Class ElevatorModelEvent is the superclass for all other event classes in our model (part 2 of 2).

Fig. 10.24 Class diagram that models the generalization between ElevatorModelEvent and its subclasses.
Event Handling

The concept of event handling in Java is similar to the concept of a collaboration described in Section 7.10. Event handling consists of an object of one class sending a particular message (which Java calls an event) to objects of other classes listening for that type of message. The difference is that the objects receiving the message must register to receive the message; therefore, event handling describes how an object sends an event to other objects “listening” for that type of event—these objects are called event listeners. To send an event, the sending object invokes a particular method of the receiving object while passing the desired event object as a parameter. In our simulation, this event object belongs to a class that extends ElevatorModelEvent.

We presented a collaboration diagram in Fig. 7.25 showing interactions of two Person objects—waitingPassenger and ridingPassenger—as they enter and exit the Elevator. Figure 10.26 shows a modified diagram that incorporates event handling. There are three differences between the diagrams. First, we provide notes—explanatory remarks about some of the graphics in the diagram. The UML represents notes as rectangles with the upper right corners “folded over.” Notes in the UML are similar to comments in Java. A dotted line associates a note with any component of the UML (object, class, arrow, etc.). In this diagram, the <<parameter>> notation specifies that the note contains the parameters of a given message: all doorOpened events pass a DoorEvent object as a parameter; all elevatorArrived events pass an ElevatorMoveEvent object; all openDoor messages pass a Location object.

1. Technically, one object sends a notification of an event—or some triggering action—to another object. However, Java parlance refers to sending this notification as “sending an event.”

<table>
<thead>
<tr>
<th>Event</th>
<th>Sent when (triggering action)</th>
<th>Sent by object of class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BellEvent</td>
<td>the Bell has rung</td>
<td>Bell</td>
</tr>
<tr>
<td>ButtonEvent</td>
<td>a Button has been pressed</td>
<td>Button</td>
</tr>
<tr>
<td></td>
<td>a Button has been reset</td>
<td></td>
</tr>
<tr>
<td>DoorEvent</td>
<td>a Door has opened</td>
<td>Door</td>
</tr>
<tr>
<td></td>
<td>a Door has closed</td>
<td></td>
</tr>
<tr>
<td>LightEvent</td>
<td>a Light has turned on</td>
<td>Light</td>
</tr>
<tr>
<td></td>
<td>a Light has turned off</td>
<td></td>
</tr>
<tr>
<td>PersonMoveEvent</td>
<td>a Person has been created</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>a Person has arrived at the Elevator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a Person has entered the Elevator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a Person has exited the Elevator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a Person has pressed a Button</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a Person has exited the simulation</td>
<td></td>
</tr>
<tr>
<td>ElevatorMoveEvent</td>
<td>the Elevator has arrived at a Floor</td>
<td>Elevator</td>
</tr>
<tr>
<td></td>
<td>the Elevator has departed from a Floor</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10.25 Triggering actions of the ElevatorModelEvent subclass events.
The second difference between the diagrams is that the interactions of Fig. 10.26 occur on the first **Floor**. This allows us to name all **Button** and **Door** objects (**first.FloorDoor** and **firstFloorButton**) to eliminate ambiguity, because the **Button** and **Door** classes each have three objects in our simulation. The interactions that occur on the second **Floor** are identical to the ones that occur on the first **Floor**.

The most substantial difference between Fig. 10.26 and Fig. 7.25 is that the **Elevator** informs objects (via an event) of an action that has already happened—the **Elevator has arrived**. The objects that receive the event then perform some action in response to the type of message they receive.

According to messages **1**, **2**, **3** and **4**, the **Elevator** performs only one action—it sends **elevatorArrived** events to objects interested in receiving those events. Specif-
ically, the Elevator object sends an ElevatorMoveEvent using the receiving object’s elevatorArrived method. Figure 10.26 begins with the Elevator sending an elevatorArrived event to the elevatorButton. The elevatorButton then resets itself (message 1.1). The Elevator then sends an elevatorArrived event to the Bell (message 2), and the Bell invokes its ringBell method, accordingly (i.e., the Bell object sends itself a ringBell message in message 2.1).

The Elevator sends an elevatorArrived message to the elevatorDoor (message 3). The elevatorDoor then opens itself by invoking its openDoor method (message 3.1). At this point, the elevatorDoor is open but has not informed the ridingPassenger of opening. Before informing the ridingPassenger, the elevatorDoor opens the firstFloorDoor by sending an openDoor message to the firstFloorDoor (message 3.2)—this guarantees that the ridingPassenger will not exit before the firstFloorDoor opens. The firstFloorDoor then informs the waitingPassenger that the firstFloorDoor has opened (message 3.2.1), and the waitingPassenger enters the Elevator (message 3.2.1.1). All messages nested in 3.2 have been passed, so the elevatorDoor may inform the ridingPassenger that elevatorDoor has opened by invoking method doorOpened of the ridingPassenger (message 3.3). The ridingPassenger responds by exiting the Elevator (message 3.3.1).2

Lastly, the Elevator informs the ElevatorShaft of the arrival (message 4). The ElevatorShaft then informs the firstFloorButton of the arrival (message 4.1), and the firstFloorButton resets itself (message 4.1.1). The ElevatorShaft then informs the firstFloorLight of the arrival (message 4.2), and the firstFloorLight illuminates itself (message 4.2.1).

**Event Listeners**

We demonstrated event handling between the Elevator and object elevatorDoor using the modified collaboration diagram of Fig. 10.26—the Elevator sends an elevatorArrived event to the elevatorDoor (message 3). We first must determine the event object that the Elevator will pass to the elevatorDoor. According to the note in the lower left-hand corner of Fig. 10.26, the Elevator passes an ElevatorMoveEvent (Fig. 10.27) object when the Elevator invokes an elevatorArrived method. The generalization diagram of Fig. 10.24 indicates that ElevatorMoveEvent is a subclass of ElevatorModelEvent, so ElevatorMoveEvent inherits the Object and Location references from ElevatorModelEvent.3

---

2. The problem of the waitingPassenger entering the Elevator (message 3.2.1.1) before the ridingPassenger has exited (message 3.3.1) remains in our collaboration diagram. We show in “Thinking About Objects” Section 15.12 how to solve this problem by using multithreading, synchronization and active classes.

3. In our simulation, all event classes have this structure—that is, the structure of class ElevatorMoveEvent is identical to the structure of class DoorEvent, ButtonEvent, etc. When you have finished reading the material in this section, we recommend that you view the implementation of the events in Appendix G to attain a better comprehension of the structure of our system events—Fig. G.1–G.7 present the code for the events, and Fig. G.8–G.14 present the code for the event listeners.
The elevatorDoor must implement an interface that “listens” for an ElevatorMoveEvent—this makes the elevatorDoor an event listener. Interface ElevatorMoveListener (Fig. 10.28) provides methods elevatorDeparted (line 8) and elevatorArrived (line 11) that enable the Elevator to notify the ElevatorMoveListener when the Elevator has arrived or departed. An interface that provides the methods for an event listener, such as ElevatorMoveListener, is called an event listener interface.

Methods elevatorArrived and elevatorDeparted each receive an ElevatorMoveEvent (Fig. 10.27) object as an argument. Therefore, when the Elevator “sends an elevatorArrived event” to another object, the Elevator passes an ElevatorMoveEvent object as an argument to the receiving object’s elevatorArrived method. We implement class Door—the class of which the elevatorDoor is an instance—in Appendix H, after we continue refining our design and learning more Java capabilities.

```java
// ElevatorMoveListener.java
package com.deitel.jhtp4.elevator.event;

public interface ElevatorMoveListener {
    // invoked when Elevator has departed
    public void elevatorDeparted( ElevatorMoveEvent moveEvent );

    // invoked when Elevator has arrived
    public void elevatorArrived( ElevatorMoveEvent moveEvent );
}
```

Fig. 10.28 Interface ElevatorMoveListener provides the methods required to listen for Elevator departure and arrival events.
Class Diagram Revisited

Figure 10.29 modifies the associations in the class diagram of Fig. 9.19 to include event handling. Note that like the collaboration diagram of Fig. 10.26, Fig. 10.29 indicates that an object informs, or **signals**, another object that some event has occurred. If an object receiving the event invokes a **private** method, the class diagram represents this method invocation as a **self association**—that is, the class contains an association with itself. The **Button**, **Door**, **Light** and **Bell** classes contain self associations; note that the association does not include an arrowhead indicating the direction of the association, because the class’s association is with itself. Lastly, the diagram includes an association between class **Door** and class **Person** (the **Door** informs a **Person** that the **Door** has opened), because we established the relationship between all **Door** objects and a **Person** object.

![Class Diagram of Simulator](image-url)
SUMMARY

- A character constant’s value is its integer value in the Unicode character set. Strings can include letters, digits and special characters such as +, -, *, / and $. A string in Java is an object of class String. String literals or string constants are often referred to as anonymous String objects and are written in double quotes in a program.

- Class String provides nine constructors.

- String method length returns the number of characters in a String.

- String method charAt returns the character at a specific position.

- Method equals is used to test any two objects for equality (i.e., the contents of the two objects are identical). The method returns true if the objects are equal, false otherwise. Method equals uses a lexicographical comparison for Strings.

- When primitive-data type values are compared with ==, the result is true if both values are identical. When references are compared with ==, the result is true if both references refer to the same object in memory.

- Java treats all anonymous Strings with the same contents as one anonymous String object.

- String method equalsIgnoreCase performs a case-insensitive String comparison.

- String method compareTo returns 0 if the Strings it is comparing are equal, a negative number if the String that invokes compareTo is less than the String that is passed as an argument and a positive number if the String that invokes compareTo is greater than the String that is passed as an argument. Method compareTo uses a lexicographical comparison.

- String method regionMatches compares portions of two Strings for equality.

- String method startsWith determines whether a String starts with the characters specified as an argument. String method endsWith determines whether a String ends with the characters specified as an argument.

- Method hashCode performs a hash code calculation that enables a String object to be stored in a hash table. This method is inherited from Object and overridden by String.

- String method indexOf locates the first occurrence of a character or a substring in a String. Method lastIndexOf locates the last occurrence of a character or a substring in a String.

- String method substring copies and returns part of an existing String object.

- String method concat concatenates two String objects and returns a new String object containing the characters from both original Strings.

- String method replace returns a new String object that replaces every occurrence in a String of its first character argument with its second character argument.

- String method toUpperCase returns a new String with uppercase letters in the positions where the original String had lowercase letters. Method toLowerCase returns a new String with lowercase letters in the positions where the original String had uppercase letters.

- String method trim returns a new String object in which all white-space characters (such as spaces, newlines and tabs) have been removed from the beginning or end of a String.

- String method toCharArray returns a new character array containing a copy of the characters in a String.

- String class method valueOf returns its argument converted to a string.

- The first time String method intern is invoked on a String it returns a reference to that String object. Subsequent invocations of intern on different String objects that have the same contents as the original String result in multiple references to the original String object.
• Class `StringBuffer` provides three constructors that enable `StringBuffers` to be initialized with no characters and an initial capacity of 16 characters; with no characters and an initial capacity specified in the integer argument or with a copy of the characters of the `String` argument and an initial capacity that is the number of characters in the `String` argument plus 16.

• `StringBuffer` method `length` returns the number of characters currently stored in a `StringBuffer`. Method `capacity` returns the number of characters that can be stored in a `StringBuffer` without allocating more memory.

• Method `ensureCapacity` ensures that a `StringBuffer` has a minimum capacity. Method `setLength` increases or decreases the length of a `StringBuffer`.

• `StringBuffer` method `charAt` returns the character at the specified index. Method `setCharAt` sets the character at the specified position. Method `getChars` returns a character array containing a copy of the characters in the `StringBuffer`.

• Class `StringBuffer` provides overloaded `append` methods to add primitive data-type, character array, `String` and `Object` values to the end of a `StringBuffer`.

• `StringBuffers` and the `append` methods are used by the Java compiler to implement the `+` and `+=` operators for concatenating `Strings`.

• Class `StringBuffer` provides overloaded `insert` methods to insert primitive data-type, character array, `String` and `Object` values at any position in a `StringBuffer`.

• Class `Character` provides a constructor that takes a character argument.

• `Character` method `isDefined` determines whether a character is defined in the Unicode character set. If so, the method returns `true`; otherwise, it returns `false`.

• `Character` method `isDigit` determines whether a character is a defined Unicode digit. If so, the method returns `true`; otherwise, it returns `false`.

• `Character` method `isJavaIdentifierStart` determines whether a character is a character that can be used as the first character of an identifier in Java [i.e., a letter, an underscore (_) or a dollar sign ($)]). If so, the method returns `true`; otherwise, it returns `false`.

• `Character` method `isJavaIdentifierPart` determines whether a character is a character that can be used in an identifier in Java [i.e., a digit, a letter, an underscore (_) or a dollar sign ($)]. If so, the method returns `true`; otherwise, it returns `false`. Method `isLetter` determines whether a character is a letter. If so, the method returns `true`; otherwise, it returns `false`. Method `isLetterOrDigit` determines whether a character is a letter or a digit. If so, the method returns `true`; otherwise, it returns `false`.

• `Character` method `isLowerCase` determines whether a character is a lowercase letter. If so, the method returns `true`; otherwise, `false`. `Character` method `isUpperCase` determines if a character is an uppercase letter. If so, the method returns `true`; otherwise, `false`.

• `Character` method `toUpperCase` converts a character to its uppercase equivalent. Method `toLowerCase` converts a character to its lowercase equivalent.

• `Character` method `digit` converts its character argument into an integer in the number system specified by its integer argument `radix`. Method `forDigit` converts its integer argument `digit` into a character in the number system specified by its integer argument `radix`.

• `Character` method `charValue` returns the `char` stored in a `Character` object. Method `toString` returns a `String` representation of a `Character`.

• `Character` method `hashCode` performs a hash code calculation on a `Character`.

• `StringTokenizer`’s default constructor creates a `StringTokenizer` for its `String` argument that will use the default delimiter string " \n\t\r", consisting of a space, a newline, a tab and a carriage return for tokenization.
• **StringTokenizer** method **countTokens** returns the number of tokens in a **String** to be tokenized.

• **StringTokenizer** method **hasMoreTokens** determines if there are more tokens in the **String** being tokenized.

• **StringTokenizer** method **nextToken** returns a **String** with the next token.

**TERMINOLOGY**

append method of class **StringBuffer**
appending strings to other strings

array of strings

capacity method of class **StringBuffer**

Character class
class character code
character constant
character set

charAt method of class **String**

ccharAt method of class **StringBuffer**

ccharValue method of class **Character**

ccompareTo method of class **String**
comparing strings

cconcat method of class **String**
concatenation
copying strings

countTokens method (**StringTokenizer**)
delimiter

digit method of class **Character**
enendsWith method of class **String**
equals method of class **String**
equalsIgnoreCase method of class **String**

foreDigit method of class **Character**
getchars method of class **String**
getchars method of class **StringBuffer**
hash table

hashCode method of class **Character**

hashCode method of class **String**

hasMoreTokens method

hexadecimal digits

indexOf method of class **String**

insert method of class **StringBuffer**

intern method of class **String**
isDefined method of class **Character**
isDigit method of class **Character**
isJavaIdentifierPart method

isJavaIdentifierStart method

isLetter method of class **Character**
isLetterOrDigit method of class **Character**
isLowerCase method of class **Character**

isUpperCase method of class **Character**

 lastIndexOf method of class **String**

length method of class **String**

length method of class **StringBuffer**

length of a string

literal

nextToken method of **StringTokenizer**
numeric code representation of a character

printing character

regionMatches method of class **String**

replace method of class **String**

search string

setCharAt method of class **StringBuffer**

startsWith method of class **String**

string

**String** class

string concatenation

string constant

string literal

string processing

**StringBuffer** class

**StringIndexOutOfBoundsException**

**StringTokenizer** class

substring method of **String** class

toCharArray method of class **String**
token
tokenizing strings

toLowerCase method of class **Character**
toLowerCase method of class **String**
toString method of class **Character**
toString method of class **String**
toString method of class **StringBuffer**
toUpperCase method of class **Character**
toUpperCase method of class **String**
trim method of class **String**

Unicode

valueOf method of class **String**

white-space characters

word processing
SELF-REVIEW EXERCISES

10.1 State whether each of the following is true or false. If false, explain why.
   a) When String objects are compared with ==, the result is true if the Strings contain
      the same values.
   b) A String can be modified after it is created.

10.2 For each of the following, write a single statement that performs the indicated task.
   a) Compare the string in s1 to the string in s2 for equality of contents.
   b) Append the string s2 to the string s1, using +=.
   c) Determine the length of the string in s1.

ANSWERS TO SELF-REVIEW EXERCISES

10.1 a) False. String objects that are compared with operator == are actually compared to determine
      if they are the same object in memory.
   b) False. String objects are constant and cannot be modified after they are created.
      StringBuffer objects can be modified after they are created.

10.2 a) s1.equals( s2 )
   b) s1 += s2;
   c) s1.length()

EXERCISES

Exercises 10.3 through 10.6 are reasonably challenging. Once you have done these problems, you
ought to be able to implement most popular card games easily.

10.3 Modify the program in Fig. 10.21 so that the card-dealing method deals a five-card poker
hand. Then write the following additional methods:
   a) Determine if the hand contains a pair.
   b) Determine if the hand contains two pairs.
   c) Determine if the hand contains three of a kind (e.g., three jacks).
   d) Determine if the hand contains four of a kind (e.g., four aces).
   e) Determine if the hand contains a flush (i.e., all five cards of the same suit).
   f) Determine if the hand contains a straight (i.e., five cards of consecutive face values).
   g) Determine if the hand contains a full house (i.e., two cards of one face value and three
      cards of another face value).

10.4 Use the methods developed in Exercise 10.3 to write a program that deals two five-card poker
hands, evaluates each hand and determines which is the better hand.

10.5 Modify the program developed in Exercise 10.4 so that it can simulate the dealer. The dealer’s
five-card hand is dealt “face down” so the player cannot see it. The program should then evaluate
the dealer’s hand and, based on the quality of the hand, the dealer should draw one, two or three
more cards to replace the corresponding number of unneeded cards in the original hand. The program
should then reevaluate the dealer’s hand. (Caution: This is a difficult problem!)

10.6 Modify the program developed in Exercise 10.5 so that it can handle the dealer’s hand automati-
cally, but the player is allowed to decide which cards of the player’s hand to replace. The program
should then evaluate both hands and determine who wins. Now, use this new program to play 20
games against the computer. Who wins more games, you or the computer? Have one of your friends
play 20 games against the computer. Who wins more games? Based on the results of these games,
make appropriate modifications to refine your poker-playing program. (This, too, is a difficult prob-
lem.) Play 20 more games. Does your modified program play a better game?
10.7 Write an application that uses `String` method `compareTo` to compare two strings input by the user. Output whether the first string is less than, equal to or greater than the second.

10.8 Write an application that uses `String` method `regionMatches` to compare two strings input by the user. The program should input the number of characters to be compared and the starting index of the comparison. The program should state whether the first string is less than, equal to or greater than the second string. Ignore the case of the characters when performing the comparison.

10.9 Write an application that uses random number generation to create sentences. Use four arrays of strings called `article`, `noun`, `verb`, and `preposition`. Create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 sentences and output them to a text area.

After the preceding program is written, modify the program to produce a short story consisting of several of these sentences. (How about the possibility of a random term paper writer!)

10.10 (Limericks) A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 10.9, write a Java program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

10.11 (Pig Latin) Write an application that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm:

To form a pig Latin phrase from an English language phrase, tokenize the phrase into words with an object of class `StringTokenizer`. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters “ay.” Thus, the word “jump” becomes “umpjay,” the word “the” becomes “hetay,” and the word “computer” becomes “omputercay.” Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Method `printLatinWord` should display each word. Each token returned from `nextToken` is passed to method `printLatinWord` to print the pig Latin word. Enable the user to input the sentence. Keep a running display of all the converted sentences in a text area.

10.12 Write an application that inputs a telephone number as a string in the form `(555) 555-5555`. The program should use an object of class `StringTokenizer` to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. The program should convert the area code string to `int` (remember `parseInt`!) and convert the phone number string to `long`. Both the area code and the phone number should be printed. Remember that you will have to change delimiter characters during the tokenization process.

10.13 Write an application that inputs a line of text, tokenizes the line with an object of class `StringTokenizer` and outputs the tokens in reverse order.

10.14 Use the string comparison methods discussed and the techniques for sorting arrays developed in Chapter 7 to write a program that alphabetizes a list of strings. Allow the user to enter the strings in a text field. Display the results in a text area.
10.15 Write an application that inputs text and outputs the text in uppercase and lowercase letters.

10.16 Write an application that inputs several lines of text and a search character and uses method `indexOf` to determine the number of occurrences of the character in the text.

10.17 Write an application based on the program in Exercise 10.16 that inputs several lines of text and uses `indexOf` to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array and print the values in tabular format after the totals have been determined.

10.18 Write an application that reads a series of strings and outputs only those strings beginning with the letter “b.” The results should be output to a text area.

10.19 Write an application that reads a series of strings and prints only those strings ending with the letters “ED.” The results should be output to a text area.

10.20 Write an application that inputs an integer code for a character and displays the corresponding character. Modify this program so that it generates all possible three-digit codes in the range form 000 to 255 and attempts to print the corresponding characters. Display the results in a text area.

10.21 Write your own versions of the `String` methods for searching strings.

10.22 Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word “bathe” include the commonly used words “ate,” “bat,” “bet,” “tab,” “hat,” “the” and “tea.”

**SPECIAL SECTION: ADVANCED STRING MANIPULATION EXERCISES**

The preceding exercises are keyed to the text and designed to test the reader’s understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

10.23 *(Text Analysis)* The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars believe there is substantial evidence indicating that Christopher Marlowe or other authors actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

a) Write an application that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

```
To be, or not to be: that is the question:
```

contains one “a,” two “b’s,” no “c’s,” etc.

b) Write an application that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc. appearing in the text. For example, Fig. 10.30 shows the counts for the phrase

```
Whether 'tis nobler in the mind to suffer
```
c) Write an application that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the words “to” three times, the word “be” two times, the word “or” once, etc. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

### 10.24 (Printing Dates in Various Formats)

Dates are printed in several common formats. Two of the more common formats are

- **04/25/1955**
- **April 25, 1955**

Write an application that reads a date in the first format and prints that date in the second format.

### 10.25 (Check Protection)

Computers are frequently employed in check-writing systems such as payroll and accounts payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Incorrect amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called **check protection**.

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example:

```
1,230.60 (check amount)
--------
12345678 (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces would ordinarily be left blank. For example,

---

**Fig. 10.30** Counts for the string "Whether 'tis nobler in the mind to suffer".
contains three blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert leading asterisks to protect the amount as follows:

```
***99.87
--------
12345678
```

Write an application that inputs a dollar amount to be printed on a check, then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing the amount.

10.26 (Writing the Word Equivalent of a Check Amount) Continuing the discussion of the previous exercise, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be written both in numbers and “spelled out” in words as well. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

a) Many computerized check-writing systems do not print the amount of the check in words. Perhaps the main reason for this omission is the fact that most high-level languages used in commercial applications do not contain adequate string-manipulation features. Another reason is that the logic for writing word equivalents of check amounts is somewhat involved.

b) Write an application that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 112.43 should be written as

```
ONE HUNDRED TWELVE and 43/100
```

10.27 (Morse Code) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 10.31.

Write an application that reads an English language phrase and encodes the phrase into Morse code. Also write a program that reads a phrase in Morse code and converts the phrase into the English language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

10.28 (A Metric Conversion Program) Write an application that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, etc. for the metric system and inches, quarts, pounds, etc. for the English system) and should respond to simple questions such as

"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
Your program should recognize invalid conversions. For example, the question

"How many feet in 5 kilograms?"

is not a meaningful question because "feet" is a unit of length while "kilograms" is a unit of mass.

**SPECIAL SECTION: CHALLENGING STRING MANIPULATION PROJECTS**

10.29 (Project: A Spelling Checker) Many popular word processing software packages have built-in spell checkers.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. Use a computerized dictionary (if you have access to one) as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it is because we simply do not know the correct spelling, so we make a "best guess." In some cases, it is because we transpose two letters (e.g., “defualt” instead of “default”). Sometimes we double-type a letter accidentally (e.g., “hamdy” instead of “handy”). Sometimes we type a nearby key instead of the one we intended (e.g., “biryhday” instead of “birthday”), and so on.
Design and implement a spell-checker application in Java. Your program should maintain an array `wordList` of strings. Enable the user to enter these strings. [Note: In Chapter 17, we introduce file processing. Once you have this capability, you can obtain the words for the spell checker from a computerized dictionary stored in a file.]

Your program should ask a user to enter a word. The program should then look up that word in the `wordList` array. If the word is present in the array, your program should print “Word is spelled correctly.”

If the word is not present in the array, your program should print “word is not spelled correctly.” Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word “default” is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as “edfault,” “dfeault,” “deafult,” “defalut,” and “deault.” When you find a new word that matches one in `wordList`, print that word in a message, such as “Did you mean "default?".”

Implement other tests, such as replacing each double letter with a single letter and any other tests you can develop to improve the value of your spell-checker.

10.30 (Project: A Crossword Puzzle Generator) Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is suggested here as a string-manipulation project requiring substantial sophistication and effort.

There are many issues the programmer must resolve to get even the simplest crossword puzzle-generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or should double-subscripted arrays be used?

The programmer needs a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program?

The really ambitious reader will want to generate the “clues” portion of the puzzle, in which the brief hints for each “across” word and each “down” word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.
Objectives

- To understand graphics contexts and graphics objects.
- To understand and be able to manipulate colors.
- To understand and be able to manipulate fonts.
- To use `Graphics` methods to draw lines, rectangles, rectangles with rounded corners, three-dimensional rectangles, ovals, arcs and polygons.
- To use methods of class `Graphics2D` from the Java2D API to draw lines, rectangles, rectangles with rounded corners, ellipses, arcs and general paths.
- To be able to specify `Paint` and `Stroke` characteristics of shapes displayed with `Graphics2D`.

One picture is worth ten thousand words.
Chinese proverb

*Treat nature in terms of the cylinder, the sphere, the cone, all in perspective.*
Paul Cezanne

*Nothing ever becomes real till it is experienced—even a proverb is no proverb to you till your life has illustrated it.*
John Keats

*A picture shows me at a glance what it takes dozens of pages of a book to expound.*
Ivan Sergeyevich Turgenev
11.1 Introduction

In this chapter, we overview several of Java’s capabilities for drawing two-dimensional shapes, controlling colors and controlling fonts. One of Java’s initial appeals was its support for graphics that enabled Java programmers to visually enhance their applets and applications. Java now contains many more sophisticated drawing capabilities as part of the Java2D API. This chapter begins with an introduction to many of the original drawing capabilities of Java. Next, we present several of the new and more powerful Java2D capabilities, such as controlling the style of lines used to draw shapes and controlling how shapes are filled with color and patterns.

Figure 11.1 shows a portion of the Java class hierarchy that includes several of the basic graphics classes and Java2D API classes and interfaces covered in this chapter. Class `Color` contains methods and constants for manipulating colors. Class `Font` contains methods and constants for manipulating fonts. Class `FontMetrics` contains methods for obtaining font information. Class `Polygon` contains methods for creating polygons. Class `Graphics` contains methods for drawing strings, lines, rectangles and other shapes. The bottom half of the figure lists several classes and interfaces from the Java2D API. Class `BasicStroke` helps specify the drawing characteristics of lines. Classes `GradientPaint` and `TexturePaint` help specify the characteristics for filling shapes with colors or patterns. Classes `GeneralPath`, `Arc2D`, `Ellipse2D`, `Line2D`, `Rectangle2D` and `RoundRectangle2D` define a variety of Java2D shapes.

To begin drawing in Java, we must first understand Java’s coordinate system (Figure 11.2), which is a scheme for identifying every possible point on the screen. By default, the upper-left corner of a GUI component (such as an applet or a window) has the coordinates (0, 0). A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate). The x-coordinate is the horizontal distance moving right from the upper-left corner. The y-coordinate is the vertical distance moving down from the upper-left corner. The x-axis describes every horizontal coordinate, and the y-axis describes every vertical coordinate.
Software Engineering Observation 11.1

The upper-left coordinate (0, 0) of a window is actually behind the title bar of the window. For this reason, drawing coordinates should be adjusted to draw inside the borders of the window. Class `Container` (a superclass of all windows in Java) has method `getInsets`, which returns an `Insets` object (package `java.awt`) for this purpose. An `Insets` object has four public members—`top`, `bottom`, `left` and `right`—that represent the number of pixels from each edge of the window to the drawing area for the window.

Text and shapes are displayed on the screen by specifying coordinates. Coordinate units are measured in pixels. A pixel is a display monitor’s smallest unit of resolution.
11.2 Graphics Contexts and Graphics Objects

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 applet we have seen in the text that performs drawing on the screen has used the Graphics object \( g \) (the argument to the applet’s paint method) to manage the applet’s graphics context. In this chapter, we demonstrate drawing in applications. However, every technique shown here can be used in applets.

The Graphics class is an abstract class (i.e., Graphics objects cannot be instantiated). This contributes to Java’s portability. Because drawing is performed differently on each platform that supports Java, 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. When Java is implemented on each platform, a derived class of Graphics is created that actually implements all the drawing capabilities. This implementation is hidden from us 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 (we discuss class Component in Chapter 12). Component method paint takes a Graphics object as an argument. This object is passed to the paint method by the system when a paint operation is required for a Component. The header for the paint method is

\[
\text{public void paint( Graphics g )}
\]

The Graphics object \( g \) receives a reference to an object of the system’s derived Graphics class. The preceding method header should look familiar to you—it is the same one...
we have been using in our applet classes. Actually, the Component class is an indirect base class of class JApplet—the superclass of every applet in this book. Many capabilities of class JApplet are inherited from class Component. The paint method defined in class Component does nothing by default—it must be overridden by the programmer.

The paint method is seldom called directly by the programmer because drawing graphics is an event-driven process. When an applet executes, the paint method is automatically called (after calls to the JApplet’s init and start methods). For paint to be called again, an event must occur (such as covering and uncovering the applet). Similarly, when any Component is displayed, that Component’s paint method is called.

If the programmer needs to call paint, a call is made to the Component class repaint method. Method repaint requests a call to the Component class update method as soon as possible to clear the Component’s background of any previous drawing, then update calls paint directly. The repaint method is frequently called by the programmer to force a paint operation. Method repaint should not be overridden, because it performs some system-dependent tasks. The update method is seldom called directly and sometimes overridden. Overriding the update method is useful for “smoothing” animations (i.e., reducing “flicker”) as we will discuss in Chapter 18, Multimedia. The headers for repaint and update are

```java
public void repaint()
public void update( Graphics g )
```

Method update takes a Graphics object as an argument, which is supplied automatically by the system when update is called.

In this chapter, we focus on the paint method. In the next chapter, we concentrate more on the event-driven nature of graphics and discuss the repaint and update methods in more detail. We also discuss in that chapter class JComponent—a superclass of many GUI components in package javax.swing. Subclasses of JComponent typically paint from their paintComponent methods.

11.3 Color Control

Colors enhance the appearance of a program and help convey meaning. For example, a traffic light has three different color lights—red indicates stop, yellow indicates caution and green indicates go.

Class Color defines methods and constants for manipulating colors in a Java program. The predefined color constants are summarized in Fig. 11.3, and several color methods and constructors are summarized in Fig. 11.4. Note that two of the methods in Fig. 11.4 are Graphics methods that are specific to colors.

Every color is created from a red, a green and a blue component. Together these components are called RGB values. All three RGB components can be integers in the range from 0 to 255, or all three RGB parts can be floating-point values in the range 0.0 to 1.0. The first RGB part defines the amount of red, the second defines the amount of green and the third defines the amount of blue. The larger the RGB value, the greater the amount of that particular color. Java enables the programmer to choose from $256 \times 256 \times 256$ (or approximately 16.7 million) colors. However, not all computers are capable of displaying all these colors. If this is the case, the computer will display the closest color it can.
<table>
<thead>
<tr>
<th>Color Constant</th>
<th>Color</th>
<th>RGB value</th>
</tr>
</thead>
<tbody>
<tr>
<td>public final static</td>
<td>Color orange</td>
<td>orange</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255, 200, 0</td>
</tr>
<tr>
<td>public final static</td>
<td>Color pink</td>
<td>pink</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255, 175, 175</td>
</tr>
<tr>
<td>public final static</td>
<td>Color cyan</td>
<td>cyan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, 255, 255</td>
</tr>
<tr>
<td>public final static</td>
<td>Color magenta</td>
<td>magenta</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255, 0, 255</td>
</tr>
<tr>
<td>public final static</td>
<td>Color yellow</td>
<td>yellow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255, 255, 0</td>
</tr>
<tr>
<td>public final static</td>
<td>Color black</td>
<td>black</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>public final static</td>
<td>Color white</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255, 255, 255</td>
</tr>
<tr>
<td>public final static</td>
<td>Color gray</td>
<td>gray</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128, 128, 128</td>
</tr>
<tr>
<td>public final static</td>
<td>Color lightGray</td>
<td>light gray</td>
</tr>
<tr>
<td></td>
<td></td>
<td>192, 192, 192</td>
</tr>
<tr>
<td>public final static</td>
<td>Color darkGray</td>
<td>dark gray</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64, 64, 64</td>
</tr>
<tr>
<td>public final static</td>
<td>Color red</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td></td>
<td>255, 0, 0</td>
</tr>
<tr>
<td>public final static</td>
<td>Color green</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, 255, 0</td>
</tr>
<tr>
<td>public final static</td>
<td>Color blue</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, 0, 255</td>
</tr>
</tbody>
</table>

**Fig. 11.3** Color class static constants and RGB values

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Color( int r, int g, int b )</td>
<td>Creates a color based on red, green and blue contents expressed as integers from 0 to 255.</td>
</tr>
<tr>
<td>public Color( float r, float g, float b )</td>
<td>Creates a color based on red, green and blue contents expressed as floating-point values from 0.0 to 1.0.</td>
</tr>
<tr>
<td>public int getRed()</td>
<td>Returns a value between 0 and 255 representing the red content.</td>
</tr>
<tr>
<td>public int getGreen()</td>
<td>Returns a value between 0 and 255 representing the green content.</td>
</tr>
<tr>
<td>public int getBlue()</td>
<td>Returns a value between 0 and 255 representing the blue content.</td>
</tr>
<tr>
<td>public Color getColor()</td>
<td>Returns a Color object representing the current color for the graphics context.</td>
</tr>
<tr>
<td>public void setColor( Color c )</td>
<td>Sets the current color for drawing with the graphics context.</td>
</tr>
</tbody>
</table>

**Fig. 11.4** Color methods and color-related Graphics methods
Common Programming Error 11.1

Spelling any static Color class constant with an initial capital letter is a syntax error.

Two Color constructors are shown in Fig. 11.4—one that takes three int arguments, and one that takes three float arguments, with each argument specifying the amount of red, green and blue, respectively. The int values must be between 0 and 255 and the float values must be between 0.0 and 1.0. The new Color object will have the specified amounts of red, green and blue. Color methods getRed, getGreen and getBlue return integer values from 0 to 255 representing the amount of red, green and blue, respectively. Graphics method getColor returns a Color object representing the current drawing color. Graphics method setColor sets the current drawing color.

The application of Fig. 11.5 demonstrates several methods from Fig. 11.4 by drawing filled rectangles and strings in several different colors.

```java
// Fig. 11.5: ShowColors.java
// Demonstrating Colors.

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

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

public class ShowColors extends JFrame {

    // constructor sets window's title bar string and dimensions
    public ShowColors() {
        super( "Using colors" );
        setSize( 400, 130 );
        setVisible( true );
    }

    // draw rectangles and Strings in different colors
    public void paint( Graphics g ) {
        // call superclass's paint method
        super.paint( g );

        // set new drawing color using integers
        g.setColor( new Color( 255, 0, 0 ) );
        g.fillRect( 25, 25, 100, 20 );
        g.drawString( "Current RGB: " + g.getColor(), 130, 40 );

        // set new drawing color using floats
        g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
        g.fillRect( 25, 50, 100, 20 );
        g.drawString( "Current RGB: " + g.getColor(), 130, 65 );
    }
}
```

Fig. 11.5 Demonstrating setting and getting a Color (part 1 of 2).
When the application begins execution, class `ShowColors`’s `paint` method (lines 23–49) is called to paint the window. Line 29 uses `Graphics` method `setColor` to set the current drawing color. Method `setColor` receives a `Color` object. The expression `newColor(255,0,0)` creates a new `Color` object that represents red (red value 255, and 0 for the green and blue values). Line 30 uses `Graphics` method `fillRect` to draw a filled rectangle in the current color. Method `fillRect` receives the same parameters as method `drawRect` (discussed in Chapter 3). Line 31 uses `Graphics` method `drawString` to draw a `String` in the current color. The expression `g.getColor()` retrieves the current color from the `Graphics` object. The returned `Color` is concatenated with string "Current RGB: ", resulting in an implicit call to class `Color`’s `toString` method. Notice that the `String` representation of the `Color` object contains the class name and package (java.awt.Color), and the red, green and blue values.

Lines 34–36 and lines 39–41 perform the same tasks again. Line 34 uses the `Color` constructor with three `float` arguments to create the color green (0.0f for red, 1.0f for green and 0.0f for blue). Note the syntax of the constants. The letter `f` appended to a floating-point constant indicates that the constant should be treated as type `float`. Normally, floating-point constants are treated as type `double`. 

Fig. 11.5 Demonstrating setting and getting a `Color` (part 2 of 2).
Line 39 sets the current drawing color to one of the predefined \texttt{Color} constants (\texttt{Color.blue}). Note that the new operator is not needed to create the constant. The \texttt{Color} constants are \texttt{static}, so they are defined when class \texttt{Color} is loaded into memory at execution time.

The statement at lines 47–48 demonstrates \texttt{Color} methods \texttt{getRed}, \texttt{getGreen} and \texttt{getBlue} on the predefined \texttt{Color.magenta} object.

Notice lines 56–57 in \texttt{main.JFrame} method \texttt{setDefaultCloseOperation} specifies the default action to take when the user clicks the close box on an application window. In this case, we specify \texttt{JFrame.EXIT\_ON\_CLOSE} to indicate that the program should terminate when the user clicks the close box. Other options are \texttt{DO\_NOTHING\_ON\_CLOSE} (to ignore the window-closing event), \texttt{HIDE\_ON\_CLOSE} (to hide the window, such that it can be redisplayed later) and \texttt{DISPOSE\_ON\_CLOSE} (to dispose of the window, such that it cannot be redisplayed later). From this point forward, we implement our own \texttt{WindowListener} only if the program should perform additional tasks when the user clicks the window’s close box. Otherwise, we use method \texttt{setDefaultCloseOperation} to specify that the program should terminate when the user clicks the close box.

\textbf{Software Engineering Observation 11.2}

To change the color, you must create a new \texttt{Color} object (or use one of the predefined \texttt{Color} constants); there are no set methods in class \texttt{Color} to change the characteristics of the current color.

One of the newer features of Java is the predefined GUI component \texttt{JColorChooser} (package \texttt{javax.swing}) for selecting colors. The application of Fig. 11.6 enables you to press a button to display a \texttt{JColorChooser} dialog. When you select a color and press the dialog’s \texttt{OK} button, the background color of the application window changes colors.

```java
// Fig. 11.6: ShowColors2.java
// Demonstrating JColorChooser.

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

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

public class ShowColors2 extends JFrame {
    private JButton changeColorButton;
    private Color color = Color.lightGray;
    private Container container;

    // set up GUI
    public ShowColors2() {
        super( "Using JColorChooser" );
    }
}
```

\textbf{Fig. 11.6} Demonstrating the \texttt{JColorChooser} dialog (part 1 of 3).
```java
container = getContentPane();
container.setLayout( new FlowLayout() );

// set up changeColorButton and register its event handler
changeColorButton = new JButton( "Change Color" );

changeColorButton.addActionListener(

    // anonymous inner class
    new ActionListener() {

        // display JColorChooser when user clicks button
        public void actionPerformed( ActionEvent event )
        {
            color = JColorChooser.showDialog( 
                ShowColors2.this, "Choose a color", color );

            // set default color, if no color is returned
            if ( color == null )
                color = Color.lightGray;

            // change content pane's background color
            container.setBackground( color );
        }
    }

); // end call to addActionListener

container.add( changeColorButton );

setSize( 400, 130 );
setVisible( true );

// execute application
public static void main( String args[] )
{
    ShowColors2 application = new ShowColors2();

    application.setDefaultCloseOperation( 
        JFrame.EXIT_ON_CLOSE );

} // end class ShowColors2
```

**Fig. 11.6** Demonstrating the `JColorChooser` dialog (part 2 of 3).
Lines 35–36 (from method `actionPerformed` for `changeColor`) use static method `showDialog` of class `JColorChooser` to display the color chooser dialog. This method returns the selected `Color` object (`null`, if the user presses `Cancel` or closes the dialog without pressing `OK`).

Method `showDialog` takes three arguments—a reference to its parent `Component`, a `String` to display in the title bar of the dialog and the initial selected `Color` for the dialog. The parent component is the window from which the dialog is displayed. While the color chooser dialog is on the screen, the user cannot interact with the parent component. This type of dialog is called a `modal dialog` and is discussed in Chapter 13. Notice the special syntax `ShowColors2.this` used in the preceding statement. When using an inner class, you can access the outer class object’s `this` reference by qualifying `this` with the name of the outer class and the dot (.`) operator.

After the user selects a color, lines 39–40 determine whether `color` is `null`, and, if so, set `color` to the default `Color.lightGray`. Line 43 uses method `setBackground` to change the background color of the content pane (represented by `container` in this program). Method `setBackground` is one of the many `Component` methods that can be used on most `GUI` components.

The second screen capture of Fig. 11.6 demonstrates the default `JColorChooser` dialog that allows the user to select a color from a variety of `color swatches`. Notice that there are actually three tabs across the top of the dialog—`Swatches`, `HSB` and `RGB`.
These represent three different ways to select a color. The HSB tab allows you to select a color based on *hue, saturation* and *brightness*. The RGB tab allows you to select a color by using sliders to select the red, green and blue components of the color. The HSB and RGB tabs are shown in Fig. 11.7.

### 11.4 Font Control

This section introduces methods and constants for font control. Most font methods and font constants are part of class Font. Some methods of class Font and class Graphics are summarized in Fig. 11.8.
Class **Font**'s constructor takes three arguments—the *font name*, *font style* and *font size*. The font name is any font currently supported by the system where the program is running, such as standard Java fonts **Monospaced**, **SansSerif** and **Serif**. The font style is **Font.PLAIN**, **Font.ITALIC** or **Font.BOLD** (static constants of class **Font**). Font styles can be used in combination (e.g., **Font.ITALIC + Font.BOLD**). The font size is measured in points. A **point** is 1/72 of an inch. **Graphics** method **setFont** sets the current drawing font—the font in which text will be displayed—to its **Font** argument.
Portability Tip 11.2

The number of fonts varies greatly across systems. The J2SDK guarantees that the fonts Serif, Monospaced, SansSerif, Dialog and DialogInput will be available.

Common Programming Error 11.2

Specifying a font that is not available on a system is a logic error. Java will substitute that system’s default font.

The program of Fig. 11.9 displays text in four different fonts, with each font in a different size. The program uses the Font constructor to initialize Font objects on lines 30, 35, 40 and 47 (each in a call to Graphics methodsetFont to change the drawing font). Each call to the Font constructor passes a font name (Serif, Monospaced or Sans-Serif) as a String, a font style (Font.PLAIN, Font.ITALIC or Font.BOLD) and a font size. Once Graphics methodsetFont is invoked, all text displayed following the call will appear in the new font until the font is changed. Note that line 35 changes the drawing color to red, so the next string displayed appears in red.

```java
public class Fonts extends JFrame {
    // set window's title bar and dimensions
    public Fonts()
    {
        super( "Using fonts" );
        setSize( 420, 125 );
        setVisible( true );
    }

    // display Strings in different fonts and colors
    public void paint( Graphics g )
    {
        // call superclass's paint method
        super.paint( g );

        // set current font to Serif (Times), bold, 12pt
        // and draw a string
        g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
        g.drawString( "Serif 12 point bold.", 20, 50 );
    }
}
```

Fig. 11.9 Using Graphics methodsetFont to change Fonts (part 1 of 2).
Chapter 11 Graphics and Java2D 615

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Software Engineering Observation 11.3

To change the font, you must create a new Font object; there are no set methods in class Font to change the characteristics of the current font.

Often, it is necessary to get information about the current font, such as the font name, the font style and the font size. Several Font methods used to get font information are summarized in Fig. 11.8. Method getStyle returns an integer value representing the current style. The integer value returned is either Font.PLAIN, Font.ITALIC, Font.BOLD or any combination of Font.PLAIN, Font.ITALIC and Font.BOLD.

Method getSize returns the font size in points. Method getName returns the current font name as a String. Method getFamily returns the name of the font family to which the current font belongs. The name of the font family is platform specific.

Fig. 11.9 Using Graphics method setFont to change Fonts (part 2 of 2).
Portability Tip 11.3

Java provides several standardized font names and maps these into system-specific font names for portability. This is transparent to the programmer.

Font methods are also available to test the style of the current font and are summarized in Fig. 11.8. The isPlain method returns true if the current font style is plain. The isBold method returns true if the current font style is bold. The isItalic method returns true if the current font style is italic.

Sometimes precise information about a font’s metrics must be known—such as height, descent (the amount a character dips below the baseline), ascent (the amount a character rises above the baseline) and leading (the difference between the descent of one line of text and the ascent of the line of text below it—i.e., the interline spacing). Figure 11.10 illustrates some of the common font metrics. Note that the coordinate passed to drawString corresponds to the lower-left corner of the baseline of the font.

Class FontMetrics defines several methods for obtaining font metrics. These methods and Graphics method getFontMetrics are summarized in Fig. 11.11.

---

### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public int getAscent() // FontMetrics class</td>
<td>Returns a value representing the ascent of a font in points.</td>
</tr>
<tr>
<td>public int getDescent() // FontMetrics class</td>
<td>Returns a value representing the descent of a font in points.</td>
</tr>
<tr>
<td>public int getLeading() // FontMetrics class</td>
<td>Returns a value representing the leading of a font in points.</td>
</tr>
<tr>
<td>public int getHeight() // FontMetrics class</td>
<td>Returns a value representing the height of a font in points.</td>
</tr>
<tr>
<td>public FontMetrics getFontMetrics() // Graphics class</td>
<td>Returns the FontMetrics object for the current drawing Font.</td>
</tr>
<tr>
<td>public FontMetrics getFontMetrics( Font f ) // Graphics class</td>
<td>Returns the FontMetrics object for the specified Font argument.</td>
</tr>
</tbody>
</table>
The program of Fig. 11.12 uses the methods of Fig. 11.11 to obtain font metric information for two fonts.

```java
// Fig. 11.12: Metrics.java
// Demonstrating methods of class FontMetrics and
// class Graphics useful for obtaining font metrics.

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

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

public class Metrics extends JFrame {

    // set window's title bar String and dimensions
    public Metrics()
    {
        super( "Demonstrating FontMetrics" );

        setSize( 510, 210 );
        setVisible( true );
    }

    // display font metrics
    public void paint( Graphics g )
    {
        // call superclass's paint method
        super.paint( g );

        g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
        FontMetrics metrics = g.getFontMetrics();
        g.drawString( "Current font: " + g.getFont(), 10, 40 );
        g.drawString( "Ascent: " + metrics.getAscent(), 10, 55 );
        g.drawString( "Descent: " + metrics.getDescent(), 10, 70 );
        g.drawString( "Height: " + metrics.getHeight(), 10, 85 );
        g.drawString( "Leading: " + metrics.getLeading(), 10, 100);

        Font font = new Font( "Serif", Font.ITALIC, 14 );
        metrics = g.getFontMetrics( font );
        g.setFont( font );
        g.drawString( "Current font: " + font, 10, 130 );
        g.drawString( "Ascent: " + metrics.getAscent(), 10, 145 );
        g.drawString( "Descent: " + metrics.getDescent(), 10, 160 );
        g.drawString( "Height: " + metrics.getHeight(), 10, 175 );
        g.drawString( "Leading: " + metrics.getLeading(), 10, 190 );
    }

    // execute application
    public static void main( String args[] )
    {
        Metrics application = new Metrics();
    }
}
```

Fig. 11.12 Obtaining font metric information (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
Line 29 creates and sets the current drawing font to a **SansSerif**, bold, 12-point font. Line 30 uses **Graphics** method `setFontMetrics` to obtain the `FontMetrics` object for the current font. Line 31 uses an implicit call to class **Font**‘s `toString` method to output the string representation of the font. Lines 32–35 use `FontMetrics` methods to obtain the ascent, descent, height and leading for the font.

Line 37 creates a new **Serif**, italic, 14-point font. Line 38 uses a second version of **Graphics** method `setFontMetrics`, which receives a **Font** argument and returns a corresponding `FontMetrics` object. Lines 41–44 obtain the ascent, descent, height and leading for the font. Notice that the font metrics are slightly different for the two fonts.

### 11.5 Drawing Lines, Rectangles and Ovals

This section presents a variety of **Graphics** methods for drawing lines, rectangles and ovals. The methods and their parameters are summarized in Fig. 11.13. For each drawing method that requires a **width** and **height** parameter, the **width** and **height** must be nonnegative values. Otherwise, the shape will not display.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public void drawLine( int x1, int y1, int x2, int y2 )</code></td>
<td>Draws a line between the point <code>(x1, y1)</code> and the point <code>(x2, y2)</code>.</td>
</tr>
<tr>
<td><code>public void drawRect( int x, int y, int width, int height )</code></td>
<td>Draws a rectangle of the specified <strong>width</strong> and <strong>height</strong>. The top-left corner of the rectangle has the coordinates <code>(x, y)</code>.</td>
</tr>
</tbody>
</table>
public void fillRect ( int x, int y, int width, int height )
Draws a solid rectangle with the specified width and height. The top-left corner of the rectangle has the coordinate (x, y).

public void clearRect ( int x, int y, int width, int height )
Draws a solid rectangle with the specified width and height in the current background color. The top-left corner of the rectangle has the coordinate (x, y).

public void drawRoundRect ( int x, int y, int width, int height, int arcWidth, int arcHeight )
Draws a rectangle with rounded corners in the current color with the specified width and height. The arcWidth and arcHeight determine the rounding of the corners (see Fig. 11.15).

public void fillRoundRect ( int x, int y, int width, int height, int arcWidth, int arcHeight )
Draws a solid rectangle with rounded corners in the current color with the specified width and height. The arcWidth and arcHeight determine the rounding of the corners (see Fig. 11.15).

public void draw3DRect ( int x, int y, int width, int height, boolean b )
Draws a three-dimensional rectangle in the current color with the specified width and height. The top-left corner of the rectangle has the coordinate (x, y). The rectangle appears raised when b is true and is lowered when b is false.

public void fill3DRect ( int x, int y, int width, int height, boolean b )
Draws a filled three-dimensional rectangle in the current color with the specified width and height. The top-left corner of the rectangle has the coordinates (x, y). The rectangle appears raised when b is true and is lowered when b is false.

public void drawOval ( int x, int y, int width, int height )
Draws an oval in the current color with the specified width and height. The bounding rectangle’s top-left corner is at the coordinates (x, y). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 11.16).

public void fillOval ( int x, int y, int width, int height )
Draws a filled oval in the current color with the specified width and height. The bounding rectangle’s top-left corner is at the coordinates (x, y). The oval touches all four sides of the bounding rectangle at the center of each side (see Fig. 11.16).

Fig. 11.13 Graphics methods that draw lines, rectangles and ovals (part 2 of 2).
The application of Fig. 11.14 demonstrates drawing a variety of lines, rectangles, 3D rectangles, rounded rectangles and ovals.

```java
// Fig. 11.14: LinesRectsOvals.java
// Drawing lines, rectangles and ovals

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

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

public class LinesRectsOvals extends JFrame {

    public LinesRectsOvals() {
        super( "Drawing lines, rectangles and ovals" );
        setSize( 400, 165 );
        setVisible( true );
    }

    public void paint( Graphics g ) {
        // call superclass's paint method
        super.paint( g );

        g.setColor( Color.red );
        g.drawLine( 5, 30, 350, 30 );

        g.setColor( Color.blue );
        g.drawRect( 5, 40, 90, 55 );
        g.fillRect( 100, 40, 90, 55 );

        g.setColor( Color.cyan );
        g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
        g.drawRoundRect( 290, 40, 90, 55, 20, 20 );

        g.setColor( Color.yellow );
        g.draw3DRect( 5, 100, 90, 55, true );
        g.fill3DRect( 100, 100, 90, 55, false );

        g.setColor( Color.magenta );
        g.drawOval( 195, 100, 90, 55 );
        g.fillOval( 290, 100, 90, 55 );
    }
}
```

Fig. 11.14 Demonstrating `Graphics` method `drawLine` (part 1 of 2).
Methods `fillRoundRect` (line 36) and `drawRoundRect` (line 37) draw rectangles with rounded corners. Their first two arguments specify the coordinates of the upper-left corner of the bounding rectangle—the area in which the rounded rectangle will be drawn. Note that the upper-left corner coordinates are not the edge of the rounded rectangle, but the coordinates where the edge would be if the rectangle had square corners. The third and fourth arguments specify the width and height of the rectangle. Their last two arguments—arcWidth and arcHeight—determine the horizontal and vertical diameters of the arcs used to represent the corners.

Methods `draw3DRect` (line 40) and `fill3DRect` (line 41) take the same arguments. The first two arguments specify the top-left corner of the rectangle. The next two arguments specify the width and height of the rectangle, respectively. The last argument determines whether the rectangle is raised (true) or lowered (false). The three-dimensional effect of `draw3DRect` appears as two edges of the rectangle in the original color and two edges in a slightly darker color. The three-dimensional effect of `fill3DRect` appears as two edges of the rectangle in the original drawing color and the fill and other two edges in a slightly darker color. Raised rectangles have the original drawing color edges at the top and left of the rectangle. Lowered rectangles have the original drawing color edges at the bottom and right of the rectangle. The three-dimensional effect is difficult to see in some colors.

Figure 11.15 labels the arc width, arc height, width and height of a rounded rectangle. Using the same value for arcWidth and arcHeight produces a quarter circle at each corner. When width, height, arcWidth and arcHeight have the same values, the result is a circle. If the values for width and height are the same and the values of arcWidth and arcHeight are 0, the result is a square.
The `drawOval` and `fillOval` methods take the same four arguments. The first two arguments specify the top-left coordinate of the bounding rectangle that contains the oval. The last two arguments specify the **width** and **height** of the bounding rectangle, respectively. Figure 11.16 shows an oval bounded by a rectangle. Note that the oval touches the center of all four sides of the bounding rectangle (the bounding rectangle is not displayed on the screen).

### 11.6 Drawing Arcs

An **arc** is a portion of a oval. Arc angles are measured in degrees. Arcs **sweep** from a **starting angle** the number of degrees specified by their **arc angle**. The starting angle indicates in degrees where the arc begins. The arc angle specifies the total number of degrees through which the arc sweeps. Figure 11.17 illustrates two arcs. The left set of axes shows an arc sweeping from zero degrees to approximately 110 degrees. Arcs that sweep in a counterclockwise direction are measured in **positive degrees**. The right set of axes shows an arc sweeping from zero degrees to approximately –110 degrees. Arcs that sweep in a clockwise direction are measured in **negative degrees**. Notice the dashed boxes around the arcs in Fig. 11.17. When drawing an arc, we specify a bounding rectangle for an oval. The arc will sweep along part of the oval. The **Graphics** methods **drawArc** and **fillArc** for drawing arcs are summarized in Fig. 11.18.
The program of Fig. 11.19 demonstrates the arc methods of Fig. 11.18. The program draws six arcs (three unfilled and three filled). To illustrate the bounding rectangle that helps determine where the arc appears, the first three arcs are displayed inside a yellow rectangle that has the same \textit{x}, \textit{y}, \textit{width} and \textit{height} arguments as the arcs.

### Table of Arc Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public void drawArc( int x, int y, int width, int height, int startAngle, int arcAngle )</code></td>
<td>Draws an arc relative to the bounding rectangle’s top-left coordinates ((x, y)) with the specified \textit{width} and \textit{height}. The arc segment is drawn starting at startAngle and sweeps arcAngle degrees.</td>
</tr>
<tr>
<td><code>public void fillArc( int x, int y, int width, int height, int startAngle, int arcAngle )</code></td>
<td>Draws a solid arc (i.e., a sector) relative to the bounding rectangle’s top-left coordinates ((x, y)) with the specified \textit{width} and \textit{height}. The arc segment is drawn starting at startAngle and sweeps arcAngle degrees.</td>
</tr>
</tbody>
</table>

### Figures

**Fig. 11.17** Positive and negative arc angles.

**Fig. 11.18** \textbf{Graphics} methods for drawing arcs.

The program of Fig. 11.19 demonstrates the arc methods of Fig. 11.18. The program draws six arcs (three unfilled and three filled). To illustrate the bounding rectangle that helps determine where the arc appears, the first three arcs are displayed inside a yellow rectangle that has the same \textit{x}, \textit{y}, \textit{width} and \textit{height} arguments as the arcs.

```java
// Fig. 11.19: DrawArcs.java
// Drawing arcs

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

public class DrawArcs extends JFrame {

    // Fig. 11.19: Demonstrating \texttt{drawArc} and \texttt{fillArc} (part 1 of 3).
```
// set window's title bar String and dimensions
public DrawArcs()
{
    super( "Drawing Arcs" );
    setSize( 300, 170 );
    setVisible( true );
}

// draw rectangles and arcs
public void paint( Graphics g )
{
    // call superclass's paint method
    super.paint( g );

    // start at 0 and sweep 360 degrees
    g.setColor( Color.yellow );
    g.drawRect( 15, 35, 80, 80 );
    g.setColor( Color.black );
    g.drawArc( 15, 35, 80, 80, 0, 360 );

    // start at 0 and sweep 110 degrees
    g.setColor( Color.yellow );
    g.drawRect( 100, 35, 80, 80 );
    g.setColor( Color.black );
    g.drawArc( 100, 35, 80, 80, 0, 110 );

    // start at 0 and sweep -270 degrees
    g.setColor( Color.yellow );
    g.drawRect( 185, 35, 80, 80 );
    g.setColor( Color.black );
    g.drawArc( 185, 35, 80, 80, 0, -270 );

    // start at 0 and sweep 360 degrees
    g.fillArc( 15, 120, 80, 40, 0, 360 );

    // start at 270 and sweep -90 degrees
    g.fillArc( 100, 120, 80, 40, 270, -90 );

    // start at 0 and sweep -270 degrees
    g.fillArc( 185, 120, 80, 40, 0, -270 );
}

// execute application
public static void main( String args[] )
{
    DrawArcs application = new DrawArcs();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

}  // end class DrawArcs

Fig. 11.19 Demonstrating drawArc and fillArc (part 2 of 3).
11.7 Drawing Polygons and Polylines

Polygons are multisided shapes. Polylines are a series of connected points. Graphics methods for drawing polygons and polylines are discussed in Fig. 11.20. Note that some methods require a Polygon object (package java.awt). Class Polygon’s constructors are also described in Fig. 11.20.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void drawPolygon( int xPoints[], int yPoints[], int points )</td>
<td></td>
</tr>
<tr>
<td>Draws a polygon. The x-coordinate of each point is specified in the xPoints array and the y-coordinate of each point is specified in the yPoints array. The last argument specifies the number of points. This method draws a closed polygon—even if the last point is different from the first point.</td>
<td></td>
</tr>
<tr>
<td>public void drawPolyline( int xPoints[], int yPoints[], int points )</td>
<td></td>
</tr>
<tr>
<td>Draws a series of connected lines. The x-coordinate of each point is specified in the xPoints array and the y-coordinate of each point is specified in the yPoints array. The last argument specifies the number of points. If the last point is different from the first point, the polyline is not closed.</td>
<td></td>
</tr>
<tr>
<td>public void drawPolygon( Polygon p )</td>
<td></td>
</tr>
<tr>
<td>Draws the specified closed polygon.</td>
<td></td>
</tr>
<tr>
<td>public void fillPolygon( int xPoints[], int yPoints[], int points )</td>
<td></td>
</tr>
<tr>
<td>Draws a solid polygon. The x-coordinate of each point is specified in the xPoints array and the y-coordinate of each point is specified in the yPoints array. The last argument specifies the number of points. This method draws a closed polygon—even if the last point is different from the first point.</td>
<td></td>
</tr>
<tr>
<td>public void fillPolygon( Polygon p )</td>
<td></td>
</tr>
<tr>
<td>Draws the specified solid polygon. The polygon is closed.</td>
<td></td>
</tr>
</tbody>
</table>
The program of Fig. 11.21 draws polygons and polylines, using the methods and constructors in Fig. 11.20.

Fig. 11.21 Demonstrating `drawPolygon` and `fillPolygon`.

```
// Fig. 11.21: DrawPolygons.java
// Drawing polygons

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

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

public class DrawPolygons extends JFrame {

    // set window's title bar String and dimensions
    public DrawPolygons()
    {
        super( "Drawing Polygons" );

        setSize( 275, 230 );
        setVisible( true );
    }

    // draw polygons and polylines
    public void paint( Graphics g )
    {
        // call superclass's paint method
        super.paint( g );

        int xValues[] = { 20, 40, 50, 30, 20, 15 };
        int yValues[] = { 50, 50, 60, 80, 80, 60 };
        Polygon polygon1 = new Polygon( xValues, yValues, 6 );

        g.drawPolygon( polygon1 );
    }
}
```
Lines 28–29 create two `int` arrays and use them to specify the points for `Polygon polygon1`. The `Polygon` constructor call at line 30 receives array `xValues`, which contains the x-coordinate of each point, array `yValues`, which contains the y-coordinate of each point, and 6 (the number of points in the polygon). Line 32 displays `polygon1` by passing it as an argument to `Graphics` method `drawPolygon`.

Lines 34–35 create two `int` arrays and use them to specify the points for a series of connected lines. Array `xValues2` contains the x-coordinate of each point and array `yValues2` contains the y-coordinate of each point. Lines 37 and 39 create `Polygon` object `polygon2` and add points to it using the `addPoint` method. Line 42 displays `polygon2` by passing it as an argument to `drawPolygon` method.

Fig. 11.21 Demonstrating `drawPolygon` and `fillPolygon`.
yValues2 contains the y-coordinate of each point. Line 37 uses Graphics method drawPolyline to display the series of connected lines specified with the arguments xValues2, yValues2 and 7 (the number of points).

Lines 39–40 create two int arrays and use them to specify the points of a polygon. Array xValues3 contains the x-coordinate of each point and array yValues3 contains the y-coordinate of each point. Line 42 displays a polygon by passing to Graphics method fillPolygon the two arrays (xValues3 and yValues3) and the number of points to draw (4).

**Common Programming Error 11.3**

An ArrayIndexOutOfBoundsException is thrown if the number of points specified in the third argument to method drawPolygon or method fillPolygon is greater than the number of elements in the arrays of coordinates that define the polygon to display.

Line 44 creates Polygon polygon2 with no points. Lines 45–49 use Polygon method addPoint to add pairs of x- and y-coordinates to the Polygon. Line 51 displays Polygon polygon2 by passing it to Graphics method fillPolygon.

### 11.8 The Java2D API

The new Java2D API provides advanced two-dimensional graphics capabilities for programmers who require detailed and complex graphical manipulations. The API includes features for processing line art, text and images in packages java.awt, java.awt.image, java.awt.color, java.awt.font, java.awt.geom, java.awt.print and java.awt.image.renderable. The capabilities of the API are far too broad to cover in this textbook. For an overview of the capabilities, see the Java2D demo (demonstrated in Chapter 3). In this section, we present an overview of several Java2D capabilities.

Drawing with the Java2D API is accomplished with an instance of class Graphics2D (package java.awt). Class Graphics2D is a subclass of class Graphics, so it has all the graphics capabilities demonstrated earlier in this chapter. In fact, the actual object we have used to draw in every paint method is a Graphics2D object that is passed to method paint and accessed via the superclass Graphics reference g. To access the Graphics2D capabilities, we must downcast the Graphics reference passed to paint to a Graphics2D reference with a statement such as

```java
Graphics2D g2d = ( Graphics2D ) g;
```

The programs of the next several sections use this technique.

### 11.9 Java2D Shapes

Next, we present several Java2D shapes from package java.awt.geom, including Ellipse2D.Double, Rectangle2D.Double, RoundRectangle2D.Double, Arc2D.Double and Line2D.Double. Note the syntax of each class name. Each of these classes represents a shape with dimensions specified as double-precision floating-point values. There is a separate version of each represented with single-precision floating-point values (such as Ellipse2D.Float). In each case, Double is a static inner class of the class to the left of the dot operator (e.g., Ellipse2D). To use the static inner class, we simply qualify its name with the outer class name.
The program of Fig. 11.22 demonstrates several Java2D shapes and drawing characteristics, such as thickening lines, filling shapes with patterns and drawing dashed lines. These are just a few of the many capabilities provided by Java2D.

```java
// Fig. 11.22: Shapes.java
// Demonstrating some Java2D 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 {

    // set window's title bar String and dimensions
    public Shapes()
    {
        super( "Drawing 2D shapes" );
        setSize( 425, 160 );
        setVisible( true );
    }

    // draw shapes with Java2D API
    public void paint( Graphics g )
    {
        // call superclass's paint method
        super.paint( g );

        // create 2D by casting g to Graphics2D
        Graphics2D g2d = ( Graphics2D ) g;

        // draw 2D ellipse filled with a blue-yellow gradient
        g2d.setPaint( new GradientPaint( 5, 30, Color.blue, 35,
                                        100, Color.yellow, true ) );
        g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );

        // draw 2D rectangle in red
        g2d.setPaint( Color.red );
        g2d.setStroke( new BasicStroke( 10.0f ) );
        g2d.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );

        // draw 2D rounded rectangle with a buffered background
        BufferedImage buffImage = new BufferedImage( 10, 10, BufferedImage.TYPE_INT_RGB );
        Graphics2D gg = buffImage.createGraphics();
        gg.setColor( Color.yellow );
        gg.fillRect( 0, 0, 10, 10 );
        gg.setColor( Color.red );
        gg.roundRect( 1, 1, 9, 9, 5 );
        BufferedImageOp renderedImage = new BufferedImageOp( new CropOp( new Rectangle2D.Double( 0, 0, 10, 10 ) ) );
        buffImage = buffImage.createImageRendered(RenderedImage.TYPE_INT_RGB, renderedImage);
        g2d.drawImage( buffImage, 105, 5, null );
        g2d.drawImage( buffImage, 5, 105, null );
        g2d.drawImage( buffImage, 105, 105, null );
        g2d.drawImage( buffImage, 5, 5, null );
        g2d.setStroke( new BasicStroke( 10.0f ) );
        g2d.draw( new RoundRectangle2D.Double( 80, 30, 65, 100, 5 ) );

    }

    public static void main( String[] args )
    {
        // create a Shapes instance
        Shapes applet = new Shapes();
        applet.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        applet.setVisible( true );
    }
}
```

Fig. 11.22 Demonstrating some Java2D shapes.
gg.fillRect( 0, 0, 10, 10 ); // draw a filled rectangle

gg.setColor( Color.black );  // draw in black

gg.drawRect( 1, 1, 6, 6 );   // draw a rectangle

gg.setColor( Color.blue );   // draw in blue

gg.fillRect( 1, 1, 3, 3 );   // draw a filled rectangle

gg.setColor( Color.red );    // draw in red

gg.fillRect( 4, 4, 3, 3 );   // draw a filled rectangle

// paint buffImage onto the JFrame

g2d.setPaint( new TexturePaint(
    buffImage, new Rectangle( 10, 10 ) ) );

g2d.fill( new RoundRectangle2D.Double(
    155, 30, 75, 100, 50, 50 ) );

// draw 2D pie-shaped arc in white

g2d.setPaint( Color.white );

g2d.setStroke( new BasicStroke( 6.0f ) );

g2d.draw( new Arc2D.Double(
    240, 30, 75, 100, 0, 270, Arc2D.PIE ) );

// draw 2D lines in green and yellow

g2d.setPaint( Color.green );

g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );

float dashes[] = { 10 };

g2d.setPaint( Color.yellow );

g2d.setStroke( new BasicStroke( 4, BasicStroke.CAP_ROUND,
    BasicStroke.JOIN_ROUND, 10, dashes, 0 ) );

g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );

// execute application

public static void main( String args[] )
{
    Shapes application = new Shapes();

    application.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE );

}

} // end class Shapes

Fig. 11.22 Demonstrating some Java2D shapes.
Line 31 casts the Graphics reference received by paint to a Graphics2D reference and assigns it to g2d to allow access to the Java2D features.

The first shape we draw is an oval filled with gradually changing colors. Lines 34–35 invoke Graphics2D method setPaint 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 introduced in Section 11.3 (class Color implements Paint), or the Paint object can be an instance of the Java2D API’s GradientPaint, SystemColor or TexturePaint classes. In this case, we use a GradientPaint object.

Class GradientPaint helps draw a shape in a gradually changing colors—called 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 if the gradient is cyclic (true) or acyclic (false). The two coordinates determine the direction of the gradient. Because the second coordinate (35, 100) is down and to the right of the first coordinate (5, 30), the gradient goes down and to the right at an angle. Because 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 36 uses Graphics2D method fill 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 specifying the bounding rectangle for the ellipse to display.

Next we draw a red rectangle with a thick border. Line 39 uses setPaint to set the Paint object to Color.red. Line 40 uses Graphics2D method setStroke to set the characteristics of the rectangle’s border (or the lines for any other shape). 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 width of the line, 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.

Line 41 uses Graphics2D method draw 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.

Next we draw a rounded rectangle filled with a pattern created in a BufferedImage (package java.awt.image) object. Lines 44–45 create the BufferedImage object. Class BufferedImage can be used to 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 indicates that the image is stored in color using the RGB color scheme.
To create the fill pattern for the rounded rectangle, we must first draw into the `BufferedImage`. Line 47 creates a `Graphics2D` object that can be used to draw into the `BufferedImage`. Lines 48–55 use methods `setColor`, `fillRect` and `drawRect` (discussed earlier in this chapter) to create the pattern.

Lines 58–59 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 60–61 use `Graphics2D` method `fill` to draw a filled `Shape` object—in this case, an instance of class `RoundRectangle2D.Double`. The constructor for class `RoundRectangle2D.Double` receives six arguments specifying the rectangle dimensions and the arc width and arc height used to determine the rounding of the corners.

Next we draw a pie-shaped arc with a thick white line. Line 64 sets the `Paint` object to `Color.white`. Line 65 sets the `Stroke` object to a new `BasicStroke` for a line 6 pixels wide. Lines 66–67 use `Graphics2D` method `draw` 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. The sixth argument specifies the arc angle. The last argument specifies the how the arc is closed. Constant `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 `Arc2D` provides two other `static` constants for specifying how the arc is closed. Constant `Arc2D.CHORD` draws a line from the starting point to the ending point. Constant `Arc2D.OPEN` specifies that the arc is not closed.

Finally, we draw two lines using `Line2D` objects—one solid and one dashed. Line 70 sets the `Paint` object to `Color.green`. Line 71 uses `Graphics2D` method `draw` to draw a `Shape` object—in this case, an instance of class `Line2D.Double`. The `Line2D.Double` constructor’s arguments specify starting coordinates and ending coordinates of the line.

Line 73 defines a one-element `float` array containing the value 10. This array will be used to describe the dashes in the dashed line. In this case, each dash will be 10 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 75 sets the `Paint` object to `Color.yellow`. Lines 76–77 set the `Stroke` object to a new `BasicStroke`. The line will be 4 pixels wide and will have rounded ends (`BasicStroke.CAP_ROUND`). If lines join together (as in a rectangle at the corners), the joining of the lines will be rounded (`BasicStroke.JOIN_ROUND`). The `dashes` argument specifies the dash lengths for the line. The last argument indicates the starting subscript in the `dashes` array for the first dash in the pattern. Line 78 then draws a line with the current `Stroke`.

Next we present a general path—a shape constructed from straight lines and complex curves. A general path is represented with an object of class `GeneralPath` (package `java.awt.geom`). The program of Fig. 11.23 demonstrates drawing a general path in the shape of a five-pointed star.
// Fig. 11.23: Shapes2.java
// Demonstrating a general path

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

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

public class Shapes2 extends JFrame {

    // set window's title bar String, background color
    // and dimensions
    public Shapes2()
    {
        super( "Drawing 2D Shapes" );

        getContentPane().setBackground( Color.yellow );
        setSize( 400, 400 );
        setVisible( true );
    }

    // draw general paths
    public void paint( Graphics g )
    {
        // call superclass's paint method
        super.paint( g );

        int xPoints[] =
        { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };

        int yPoints[] =
        { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };

        Graphics2D g2d = ( 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)
        g2d.translate( 200, 200 );
    }
}

Fig. 11.23  Demonstrating some Java2D shapes
Lines 31–34 define two `int` arrays representing the $x$- and $y$-coordinates of the points in the star. Line 39 defines `GeneralPath` object `star`.

![Fig. 11.23 Demonstrating some Java2D shapes](image.png)
Line 42 uses `GeneralPath` method `moveTo` to specify the first point in the `star`. The `for` structure at lines 45–46 use `GeneralPath` method `lineTo` 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 49 uses `GeneralPath` method `closePath` to draw a line from the last point to the point specified in the last call to `moveTo`. This completes the general path.

Line 52 uses `Graphics2D` method `translate` to move the drawing origin to location (200, 200). All drawing operations now use location (200, 200) as (0, 0).

The `for` structure at line 55–68 draws the `star` 20 times by rotating it around the new origin point. Line 58 uses `Graphics2D` method `rotate` to rotate the next displayed shape. The argument specifies the rotation angle in radians (with 360° = 2\(\pi\) radians). Line 67 uses `Graphics2D` method `fill` to draw a filled version of the `star`.

### 11.10 (Optional Case Study) Thinking About Objects: Designing Interfaces with the UML

In Section 10.22, we incorporated event handling into our simulation by modifying the collaboration diagram that deals with passengers entering and exiting the elevator. We included both event handling and inheritance in that diagram. The `Elevator` informs its `Door` of the `Elevator`’s arrival. This `Door` opens the arrival `Floor`’s `Door` by obtaining its handle through a `Location` object (which was included in the arrival event), and potentially two `Person` objects exit and enter the `Elevator` after both `Doors` open. We also discussed listener interfaces. In this section, we represent our listener interface with the UML.

#### Realizations

The UML expresses the relationship between a class and an interface through a *realization*. A class realizes, or implements, the behaviors of an interface. A class diagram can show a realization between classes and interfaces. As mentioned in “Thinking About Objects” Section 3.8, the UML provides two notations to draw a class diagram—the complete diagram and the elided (condensed) diagram. Figure 11.24 shows the complete class diagram that models the realization between class `Person` and interface `DoorListener`. The diagram is similar to the generalization diagram, except that the arrow expressing the relationship is dashed instead of solid. Note that the middle compartment in interface `DoorListener` is empty, because interfaces do not contain variables—interfaces can contain constants, but interface `DoorListener` does not contain any constants. Lastly, note the word *interface* placed in guillemets (« ») located in the first compartment of interface `DoorListener`. This notation distinguishes interface `DoorListener` as an interface in our system. Items placed in guillemets are called *stereotypes* in the UML. A stereotype indicates an element’s role—or purpose—in a UML diagram.

Figure 11.25 shows the alternate way to represent the realization of class `Person` and interface `DoorListener` in the UML. Figure 11.25 is the elided diagram of Fig. 11.24. The small circle represents the interface, and the solid line represents the realization. By hiding its operations, we condense the interface, making it easier to read; however, in doing so, we sacrifice the information about its behaviors. When constructing an elided diagram, common practice is to place the information regarding any behavior in a separate diagram—for example, we place the full `DoorListener` class in the class diagram of Fig. 11.28.
Forward engineering from the UML to implemented Java code benefits from well-constructed realization diagrams. When declaring any class, specify the realization between that class and its interface—that class will “implement” the interface and override the interface’s methods. For example, we use Fig. 11.24 to begin constructing `Person.java`:

```java
public class Person implements DoorListener {
    // constructor
    public Person() {}

    // methods of DoorListener
    public void doorOpened( DoorEvent doorEvent ) {}
    public void doorClosed( DoorEvent doorEvent ) {}
}
```

Figure 11.26 shows the Java complete implementation for Fig. 11.24. Lines 6–8 and lines 14–15 include the attributes and operations of `Person`, respectively—in this case, the `doorOpened` operation (line 14) was already included when we implemented the `DoorListener` interface, so we include only the attributes of `Person`:

```java
1  // Person.java
2  // Generated from Fig. 11.24
3  public class Person implements DoorListener {
```
When a Door opens or closes, that Door invokes only those methods declared in interface DoorListener, but only if the Person has registered with that Door to receive DoorEvents. Finally, we present the elided class diagram that models the realizations in our elevator model in Fig. 11.27—the elided diagram does not contain any interface methods (making the diagram easier to read), so we present the class diagram for interfaces in Fig. 11.28, which shows all interface methods. Refer to these diagrams when studying the elevator simulation implementation in Appendices G, H and I.

```java
// attributes
private int ID;
private boolean moving = true;
private Location location;

// constructor
public Person() {}

// methods of DoorListener
public void doorOpened( DoorEvent doorEvent ) {}
public void doorClosed( DoorEvent doorEvent ) {}
```
According to Fig. 11.27, classes **Door**, **Light**, **Bell** and **Button** implement interface **ElevatorMoveListener**. Class **Elevator** implements interfaces **ButtonListener**, **DoorListener** and **BellListener**. Class **ElevatorModel** implements interface **PersonMoveListener**. Class **ElevatorShaft** implements interfaces **LightListener**, **ButtonListener** and **DoorListener**. Lastly, class
Person implements interface DoorListener. We reexamine Fig. 11.27 in Appendix H when we begin coding our model.

In this section we showed how to represent interfaces and realizations with the UML. We also presented class diagrams showing the listener interfaces and their realizations for our elevator simulation. In “Thinking About Objects” Section 12.16, we model how the user interacts with our simulation.

**SUMMARY**

- A coordinate system is a scheme for identifying every possible point on the screen.
- The upper-left corner of a GUI component has the coordinates $(0, 0)$. A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate).
- Coordinate units are measured in pixels. A pixel is a display monitor’s smallest unit of resolution.
- A graphics context enables drawing on the screen in Java. A Graphics object manages a graphics context by controlling how information is drawn.
- Graphics objects contain methods for drawing, font manipulation, color manipulation and so on.
- Method paint is normally called in response to an event, such as uncovering a window.
- Method repaint requests a call to Component method update as soon as possible to clear the Component’s background of any previous drawing, then update calls paint directly.
- Class Color defines methods and constants for manipulating colors in a Java program.
- Java uses RGB colors in which the red, green and blue color components are integers in the range from 0 to 255 or floating-point values in the range from 0.0 to 1.0. The larger the RGB value, the greater the amount of that particular color.
- Color methods getRed, getGreen and getBlue return integer values from 0 to 255 representing the amount of red, green and blue in a Color.
- Class Color provides 13 predefined Color objects.
- Java provides class JColorChooser to display a dialog for selecting colors.
- static method showDialog of class JColorChooser displays a color chooser dialog. This method returns the selected Color object (null, if none is selected).
- The default JColorChooser dialog allows you to select a color from a variety of color swatches. The HSB tab allows you to select a color based on hue, saturation and brightness. The RGB tab allows you to select a color by using sliders for the red, green and blue components of the color.
- Component method setBackground (one of the many Component methods that can be used on most GUI components) changes the background color of a component.
- Class Font’s constructor takes three arguments—the font name, the font style and the font size. The font name is any font currently supported by the system. The font style is Font.PLAIN, Font.ITALIC or Font.BOLD. The font size is measured in points.
- Graphics method setFont sets the drawing font.
- Class FontMetrics defines several methods for obtaining font metrics.
- Graphics method getFontMetrics with no arguments obtains the FontMetrics object for the current font. Graphics method getFontMetrics that receives a Font argument returns a corresponding FontMetrics object.
• Methods `draw3DRect` and `fill3DRect` take five arguments specifying the top-left corner of the rectangle, the `width` and `height` of the rectangle and whether the rectangle is raised (`true`) or lowered (`false`).

• Methods `drawRoundRect` and `fillRoundRect` draw rectangles with rounded corners. Their first two arguments specify the upper-left corner, the third and fourth arguments specify the `width` and `height`, and the last two arguments—`arcWidth` and `arcHeight`—determine the horizontal and vertical diameters of the arcs used to represent the corners.

• Methods `drawOval` and `fillOval` take the same arguments—the top-left coordinate and the `width` and the `height` of the bounding rectangle that contains the oval.

• An arc is a portion of an oval. Arcs sweep from a starting angle the number of degrees specified by their arc angle. The starting angle specifies where the arc begins and the arc angle specifies the total number of degrees through which the arc sweeps. Arcs that sweep counterclockwise are measured in positive degrees and arcs that sweep clockwise are measured in negative degrees.

• Methods `drawArc` and `fillArc` take the same arguments—the top-left coordinate, the `width` and the `height` of the bounding rectangle that contains the arc and the `startAngle` and `arc-angle` that define the sweep of the arc.

• Polygons are multisided shapes. Polylines are a series of connected points.

• One `Polygon` constructor receives an array containing the x-coordinate of each point, an array containing the y-coordinate of each point and the number of points in the polygon.

• One version of `Graphics` method `drawPolygon` displays a `Polygon` object. Another version receives an array containing the x-coordinate of each point, an array containing the y-coordinate of each point and the number of points in the polygon and displays the corresponding polygon.

• `Graphics` method `drawPolyline` displays a series of connected lines specified by its arguments (an array containing the x-coordinate of each point, an array containing the y-coordinate of each point and the number of points).

• `Polygon` method `addPoint` adds pairs of x- and y-coordinates to a `Polygon`.

• The Java2D API provides advanced two-dimensional graphics capabilities for processing line art, text and images.

• To access the `Graphics2D` capabilities, downcast the `Graphics` reference passed to `paint` to a `Graphics2d` reference.

• `Graphics2D` method `setPaint` sets the `Paint` object that determines the color and texture 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 Java2D API’s `GradientPaint`, `SystemColor` or `TexturePaint` classes.

• Class `GradientPaint` draws a shape in a gradually changing color called a gradient.

• `Graphics2D` method `fill` draws a filled `Shape` object. The `Shape` object is an instance of any class that implements interface `Shape`.

• The `Ellipse2D.Double` constructor receives four arguments specifying the bounding rectangle for the ellipse to display.

• `Graphics2D` method `setStroke` sets the characteristics of the lines used to draw a shape. Method `setStroke` requires a `Stroke` object as its argument. The `Stroke` object is an instance of any class that implements interface `Stroke`, such as a `BasicStroke`.

• `Graphics2D` method `draw` draws a `Shape` object. The `Shape` object is an instance of any class that implements interface `Shape`.

• The `Rectangle2D.Double` constructor receives four arguments specifying the upper-left x-coordinate, upper-left y-coordinate, width and height of the rectangle.
• Class `BufferedImage` can be used to produce images in color and gray scale.

• A `TexturePaint` object uses the image stored in its associated `BufferedImage` as the fill texture for a filled-in shape.

• The `RoundRectangle2D.Double` constructor receives six arguments specifying the rectangle’s dimensions and the arc width and arc height used to determine the rounding of the corners.

• The `Arc2D.Double` constructor’s first four arguments specify 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. The sixth argument specifies the end angle. The last argument specifies the type of arc (`Arc2D.PIE`, `Arc2D.CHORD` or `Arc2D.OPEN`).

• The `Line2D.Double` constructor’s arguments specify starting and ending line coordinates.

• A general path is a shape constructed from straight lines and complex curves represented with an object of class `GeneralPath` (package `java.awt.geom`).

• `GeneralPath` method `moveTo` specifies the first point in a general path. `GeneralPath` method `lineTo` 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. `GeneralPath` method `closePath` draws a line from the last point to the point specified in the last call to `moveTo`.

• `Graphics2D` method `translate` moves the drawing origin to a new location. All drawing operations now use that location as $(0, 0)$.

• `Graphics2D` method `rotate` to rotate the next that is displayed. Its argument specifies the rotation angle in radians (with $360° = 2\pi$ radians).

**TERMINOLOGY**

`addPoint` method  
angle  
arcs bounded by a rectangle  
arcs height  
arcs sweeping through an angle  
arcs width  
`Arc2D.Double` class  
`ascent`  
background color  
baseline  
bounding rectangle  
`BufferedImage` class  
closed polygon  
`closePath` method  
`Color` class  
`Component` class  
coordinate  
coordinate system  
degree  
descent  
draw an arc  
`draw` method  
`draw3DRect` method  
`drawArc` method  
`drawLine` method  
`drawOval` method  
`drawPolygon` method  
`drawPolyline` method  
`drawRect` method  
`drawRoundRect` method  
`Ellipse2D.Double` class  
event  
event-driven process  
`fill` method  
`fill3DRect` method  
`fillArc` method  
filled polygon  
`fillOval` method  
`fillPolygon` method  
`fillRect` method  
`fillRoundRect` method  
font  
`Font` class  
font metrics  
font name  
font style  
`FontMetrics` class  
`GeneralPath` class  
`getAscent` method  
`getBlue` method  
`getDescent` method  
`getFamily` method
SELF-REVIEW EXERCISES

11.1 Fill in the blanks in each of the following statements:
   a) In Java2D, method _________ of class _________ sets the characteristics of a line used
to draw a shape.
   b) Class _________ helps define the fill for a shape such that the fill gradually changes
   from one color to another.
   c) The _________ method of class Graphics draws a line between two points.
   d) RGB is short for _________, _________ and _________.
   e) Font sizes are measured in units called _________.
   f) Class _________ helps define the fill for a shape using a pattern drawn in a BufferedImage.

11.2 State whether each of the following is true or false. If false, explain why.
   a) The first two arguments of Graphics method drawOval specify the center coordinate
   of the oval.
   b) In the Java coordinate system, x values increase from left to right.
   c) Method fillPolygon draws a solid polygon in the current color.
   d) Method drawArc allows negative angles.
   e) Method getSize returns the size of the current font in centimeters.
   f) Pixel coordinate (0, 0) is located at the exact center of the monitor.

11.3 Find the error(s) in each of the following and explain how to correct the error(s). Assume that
   g is a Graphics object.
a) `g.setFont( "SansSerif" );
b) `g.erase( x, y, w, h );` // clear rectangle at (x, y)
c) `Font f = new Font( "Serif", Font.BOLDITALIC, 12 );`
d) `g.setColor( Color.Yellow );` // change color to yellow

**ANSWERS TO SELF-REVIEW EXERCISES**

11.1  
a) `setStroke, Graphics2D.`  
b) `GradientPaint.`  
c) `drawLine.`  
d) red, green, blue.  
e) points.  
f) `TexturePaint.`

11.2  
a) False. The first two arguments specify the upper-left corner of the bounding rectangle.  
b) True.  
c) True.  
d) True.  
e) False. Font sizes are measured in points.  
f) False. The coordinate (0,0) corresponds to the upper-left corner of a GUI component on which drawing occurs.

11.3  
a) The `setFont` method takes a `Font` object as an argument—not a `String`.  
b) The `Graphics` class does not have an `erase` method. The `clearRect` method should be used.  
c) `Font.BOLDITALIC` is not a valid font style. To get a bold italic font, use `Font.BOLD + Font.ITALIC`.  
d) `Yellow` should begin with a lowercase letter: `g.setColor( Color.yellow );`

**EXERCISES**

11.4  
Fill in the blanks in each of the following statements:

a) Class _________ of the Java2D API is used to define ovals.
   
b) Methods `draw` and `fill` of class `Graphics2D` require an object of type _________ as their argument.
   
c) The three constants that specify font style are _________, _________ and _________.
   
d) `Graphics2D` method _________ sets the painting color for Java2D shapes.

11.5  
State whether each of the following is true or false. If false, explain why.

a) The `drawPolygon` method automatically connects the endpoints of the polygon.
   
b) The `drawLine` method draws a line between two points.
   
c) The `fillArc` method uses degrees to specify the angle.
   
d) In the Java coordinate system, `y` values increase from top to bottom.
   
e) The `Graphics` class inherits directly from class `Object`.
   
f) The `Graphics` class is an `abstract` class.
   
g) The `Font` class inherits directly from class `Graphics`.

11.6  
Write a program that draws a series of eight concentric circles. The circles should be separated by 10 pixels. Use the `drawOval` method of class `Graphics`.

11.7  
Write a program that draws a series of eight concentric circles. The circles should be separated by 10 pixels. Use the `drawArc` method.

11.8  
Modify your solution to Exercise 11.6 to draw the ovals by using instances of class `Ellipse2D.Double` and method `draw` of class `Graphics2D`.

11.9  
Write a program that draws lines of random lengths in random colors.

11.10 
Modify your solution to Exercise 11.9 to draw random lines, in random colors and random line thicknesses. Use class `Line2D.Double` and method `draw` of class `Graphics2D` to draw the lines.
11.11 Write a program that displays randomly generated triangles in different colors. Each triangle should be filled with a different color. Use class \texttt{GeneralPath} and method \texttt{fill} of class \texttt{Graphics2D} to draw the triangles.

11.12 Write a program that randomly draws characters in different font sizes and colors.

11.13 Write a program that draws an 8-by-8 grid. Use the \texttt{drawLine} method.

11.14 Modify your solution to Exercise 11.13 to draw the grid using instances of class \texttt{Line2D.Double} and method \texttt{draw} of class \texttt{Graphics2D}.

11.15 Write a program that draws a 10-by-10 grid. Use the \texttt{drawRect} method.

11.16 Modify your solution to Exercise 11.15 to draw the grid by using instances of class \texttt{Rectangle2D.Double} and method \texttt{draw} of class \texttt{Graphics2D}.

11.17 Write a program that draws a tetrahedron (a pyramid). Use class \texttt{GeneralPath} and method \texttt{draw} of class \texttt{Graphics2D}.

11.18 Write a program that draws a cube. Use class \texttt{GeneralPath} and method \texttt{draw} of class \texttt{Graphics2D}.

11.19 In Exercise 3.9, you wrote an applet that input the radius of a circle from the user and displayed the circle’s diameter, circumference and area. Modify your solution to Exercise 3.9 to read a set of coordinates in addition to the radius. Then draw the circle and display the circle’s diameter, circumference and area, using an \texttt{Ellipse2D.Double} object to represent the circle and method \texttt{draw} of class \texttt{Graphics2D} to display the circle.

11.20 Write an application that simulates a screen saver. The application should randomly draw lines using method \texttt{drawLine} of class \texttt{Graphics}. After drawing 100 lines, the application should clear itself and start drawing lines again. To allow the program to draw continuously, place a call to \texttt{repaint} as the last line in method \texttt{paint}. Do you notice any problems with this on your system?

11.21 Here is a peek ahead. Package \texttt{javax.swing} contains a class called \texttt{Timer} that is capable of calling method \texttt{actionPerformed} of interface \texttt{ActionListener} at a fixed time interval (specified in milliseconds). Modify your solution to Exercise 11.20 to remove the call to \texttt{repaint} from method \texttt{paint}. Define your class so it implements \texttt{ActionListener}. (The \texttt{actionPerformed} method should simply call \texttt{repaint}.) Define an instance variable of type \texttt{Timer} called \texttt{timer} in your class. In the constructor for your class, write the following statements:

\begin{verbatim}
    timer = new Timer( 1000, this );
    timer.start();
\end{verbatim}

This creates an instance of class \texttt{Timer} that will call \texttt{this} object’s \texttt{actionPerformed} method every 1000 milliseconds (i.e., every second).

11.22 Modify your solution to Exercise 11.21 to enable the user to enter the number of random lines that should be drawn before the application clears itself and starts drawing lines again. Use a \texttt{JTextField} to obtain the value. The user should be able to type a new number into the \texttt{JTextField} at any time during the program’s execution. Use an inner class definition to perform event handling for the \texttt{JTextField}.

11.23 Modify your solution to Exercise 11.21 such that it uses random number generation to choose different shapes to display (use methods of class \texttt{Graphics}).

11.24 Modify your solution to Exercise 11.23 to use classes and drawing capabilities of the Java2D API. For shapes such as rectangles and ellipses, draw them with randomly generated gradients (use class \texttt{GradientPaint} to generate the gradient).
11.25 Write a graphical version of your solution to Exercise 6.37—the *Towers of Hanoi*. After studying Chapter 18, you will be able to implement a version of this exercise using Java’s image, animation and audio capabilities.

11.26 Modify the die-rolling program of Fig. 7.8 so that it updates the counts for each side of the die after each roll. Convert the application into a windowed application (i.e., a subclass of `JFrame`) and use `Graphics` method `drawString` to output the totals.

11.27 Modify your solution to Exercise 7.21—*Turtle Graphics*—to add a graphical user interface using `JTextField` and `JButton`s. Also, draw lines rather than drawing asterisks (*). When the turtle graphics program specifies a move, translate the number of positions into a number of pixels on the screen by multiplying the number of positions by 10 (or any value you choose). Implement the drawing with Java2D API features.

11.28 Produce a graphical version of the Knight’s Tour problem (Exercises 7.22, 7.23 and 7.26). As each move is made, the appropriate cell of the chessboard should be updated with the proper move number. If the result of the program is a full tour or a closed tour, the program should display an appropriate message. If you would like, use class `Timer` (see Exercise 11.24) to help animate the Knight’s Tour. Every second, the next move should be made.

11.29 Produce a graphical version of the *Tortoise and the Hare* simulation (Exercise 7.41). Simulate the mountain by drawing an arc that extends from the bottom-left of the window to the top-right of the window. The tortoise and the hare should race up the mountain. Implement the graphical output so the tortoise and the hare are actually printed on the arc every move. [Note: Extend the length of the race from 70 to 300 to allow yourself a larger graphics area.]

11.30 Produce a graphical version of the *Maze Traversal* problem (Exercises 7.38-7.40). Use the mazes you produced as guides for creating the graphical versions. While the maze is being solved, a small circle should be displayed in the maze indicating the current position. If you would like, use class `Timer` (see Exercise 11.24) to help animate the traversal of the maze. Every second, the next move should be made.

11.31 Produce a graphical version of the *Bucket Sort* (Exercise 7.28) that shows each value being placed into the appropriate bucket and eventually being copied back to the original array.

11.32 Write a program that uses method `drawPolyline` to draw a spiral.

11.33 Write a program that inputs four numbers and graphs the numbers as a pie chart. Use class `Arc2D.Double` and method `fill` of class `Graphics2D` to perform the drawing. Draw each piece of the pie in a separate color.

11.34 Write an applet that inputs four numbers and graphs the numbers as a bar graph. Use class `Rectangle2D.Double` and method `fill` of class `Graphics2D` to perform the drawing. Draw each bar in a different color.
Graphical User Interface Components: Part 1

Objectives

• To understand the design principles of graphical user interfaces (GUI).
• To be able to build graphical user interfaces.
• To understand the packages containing GUI components and event-handling classes and interfaces.
• To be able to create and manipulate buttons, labels, lists, text fields and panels.
• To understand mouse events and keyboard events.
• To understand and be able to use layout managers.

… the wisest prophets make sure of the event first.
Horace Walpole

Do you think I can listen all day to such stuff?
Lewis Carroll

Speak the affirmative; emphasize your choice by utter ignoring of all that you reject.
Ralph Waldo Emerson

You pays your money and you takes your choice.
Punch

Guess if you can, choose if you dare.
Pierre Corneille

All hope abandon, ye who enter here!
Dante Alighieri

Exit, pursued by a bear.
William Shakespeare
12.1 Introduction

A graphical user interface (GUI) presents a pictorial interface to a program. A GUI (pronounced “GOO-EE”) gives a program a distinctive “look” and “feel.” Providing different programs with a consistent set of intuitive user interface components provides users with a basic level of familiarity with each program before they ever use it. In turn, this reduces the time users require to learn a program and increases their ability to use the program in a productive manner.

**Look-and-Feel Observation 12.1**

Consistent user interfaces enable a user to learn new applications faster.

As an example of a GUI, Fig. 12.1 contains a Netscape Navigator window with some of its GUI components labeled. In the window, there is a menu bar containing menus (File, Edit, View etc.). Below the menu bar there is a set of buttons that each have a defined task in Netscape Navigator. To the right of the buttons there is a text field in which the user can type the name of the World Wide Web site to visit. The menus, buttons and text fields are part of Netscape Navigator’s GUI. They enable you to interact with the Navigator program.
In this chapter and the next, we demonstrate many GUI components that enable users to interact with your programs.

GUIs are built from GUI components (sometimes called controls or widgets—short-hand notation for window gadgets). A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. Several common GUI components are listed in Figure 12.2. In the sections that follow, we discuss each of these GUI components in detail. In the next chapter, we discuss more advanced GUI components.

![Fig. 12.1 A sample Netscape Navigator window with GUI components.](image)

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JLabel</td>
<td>An area where uneditable text or icons can be displayed.</td>
</tr>
<tr>
<td>JTextField</td>
<td>An area in which the user inputs data from the keyboard. The area can also display information.</td>
</tr>
<tr>
<td>JButton</td>
<td>An area that triggers an event when clicked.</td>
</tr>
<tr>
<td>JCheckBox</td>
<td>A GUI component that is either selected or not selected.</td>
</tr>
<tr>
<td>JComboBox</td>
<td>A drop-down list of items from which the user can make a selection by clicking an item in the list or possibly by typing into the box.</td>
</tr>
<tr>
<td>JList</td>
<td>An area where a list of items is displayed from which the user can make a selection by clicking once on any element in the list. Double-clicking an element in the list generates an action event. Multiple elements can be selected.</td>
</tr>
<tr>
<td>JPanel</td>
<td>A container in which components can be placed.</td>
</tr>
</tbody>
</table>

![Fig. 12.2 Some basic GUI components.](image)
12.2 Swing Overview

The classes that create the GUI components of Fig. 12.2 are part of the Swing GUI components from package `javax.swing`. These GUI components became standard in Java with the release of the Java 2 platform version 1.2. Most Swing components (as they are commonly called) are written, manipulated and displayed completely in Java (so-called pure Java components).

The original GUI components from the Abstract Windowing Toolkit package `java.awt` (also called the AWT) are tied directly to the local platform’s graphical user interface capabilities. When a Java program with an AWT GUI executes on different Java platforms, the program’s GUI components display differently on each platform. Consider a program that displays an object of type `Button` (package `java.awt`). On a computer running the Microsoft Windows operating system, the `Button` will have the same look and feel as the buttons in other Windows applications. Similarly, on a computer running the Apple Macintosh operating system, the `Button` will have the same look and feel as the buttons in other Macintosh applications. In addition to the differences in appearance, sometimes the manner in which a user interacts with a particular AWT component differs between platforms.

Together, the appearance and how the user interacts with the program are known as that program’s look and feel. The Swing components allow the programmer to specify a uniform look and feel across all platforms. In addition, Swing enables programs to provide a custom look and feel for each platform or even to change the look and feel while the program is running. For example, a program could enable users to choose their preferred look and feel.

Look-and-Feel Observation 12.2

Swing components are written in Java, so they provide a greater level of portability and flexibility than the original Java GUI components from package `java.awt`.

Swing components are often referred to as lightweight components—they are written completely in Java so they are not “weighed down” by the complex GUI capabilities of the platform on which they are used. AWT components (many of which parallel the Swing components) that are tied to the local platform are correspondingly called heavyweight components—they rely on the local platform’s windowing system to determine their functionality and their look and feel. Each heavyweight component has a peer (from package `java.awt.peer`) that is responsible for the interactions between the component and the local platform that display and manipulate the component. Several Swing components are still heavyweight components. In particular, subclasses of `java.awt.Window` (such as `JFrame` used in several previous chapters) that display windows on the screen and subclasses of `java.applet.Applet` (such as `JApplet`) still require direct interaction with the local windowing system. As such, heavyweight Swing GUI components are less flexible than many of the lightweight components we will demonstrate.

Portability Tip 12.1

The look of a GUI defined with heavyweight GUI components from package `java.awt` may vary across platforms. Heavyweight components “tie” into the “local” platform GUI, which varies from platform to platform.

Figure 12.3 shows an inheritance hierarchy of the classes that define attributes and behaviors that are common to most Swing components. Each class is displayed with its
fully qualified package name and class name. Much of each GUI component’s functionality is derived from these classes. A class that inherits from the `Component` class is a `Component`. For example, class `Container` inherits from class `Component`, and class `Component` inherits from `Object`. Thus, a `Container` is a `Component` and is an `Object`, and a `Component` is an `Object`. A class that inherits from class `Container` is a `Container`. Thus, a `JComponent` is a `Container`.

**Software Engineering Observation 12.1**

To use GUI components effectively, the `javax.swing` and `java.awt` inheritance hierarchies must be understood—especially class `Component`, class `Container` and class `JComponent`, which define features common to most Swing components.

Class `Component` defines the common attributes and behaviors of all subclasses of `Component`. With few exceptions, most GUI components extend class `Component` directly or indirectly. One method that originates in class `Component` that has been used frequently to this point is `paint`. Other methods discussed previously that originated in `Component` are `repaint` and `update`. It is important to understand the methods of class `Component` because much of the functionality inherited by every subclass of `Component` is defined by the `Component` class originally. Operations common to most GUI components (both Swing and AWT) are found in class `Component`.

**Good Programming Practice 12.1**

Study the methods of class `Component` in the Java 2 SDK on-line documentation to learn the capabilities common to most GUI components.

A `Container` is a collection of related components. In applications with `JFrames` and in applets, we attach components to the content pane, which is an object of class `Container`. Class `Container` defines the common attributes and behaviors for all subclasses of `Container`. One method that originates in class `Container` is `add` for adding components to a `Container`. Another method that originates in class `Container` is `setLayout`, which enables a program to specify the layout manager that helps a `Container` position and size its components.

**Good Programming Practice 12.2**

Study the methods of class `Container` in the Java 2 SDK on-line documentation to learn the capabilities common to every container for GUI components.

Class `JComponent` is the superclass to most Swing components. This class defines the common attributes and behaviors of all subclasses of `JComponent`.

![Fig. 12.3 Common superclasses of many of the Swing components.](image-url)
Good Programming Practice 12.3

Study the methods of class JComponent in the Java 2 SDK on-line documentation to learn the capabilities common to every container for GUI components.

Swing components that subclass JComponent have many features, including:

1. A pluggable look and feel that can be used to customize the look and feel when the program executes on different platforms.

2. Shortcut keys (called mnemonics) for direct access to GUI components through the keyboard.

3. Common event-handling capabilities for cases where several GUI components initiate the same actions in a program.

4. Brief descriptions of a GUI component’s purpose (called tool tips) that are displayed when the mouse cursor is positioned over the component for a short time.

5. Support for assistive technologies such as braille screen readers for blind people.

6. Support for user interface localization—customizing the user interface for display in different languages and cultural conventions.

These are just some of the many features of the Swing components. We discuss several of these features here and in Chapter 13.

12.3 JLabel

Labels provide text instructions or information on a GUI. Labels are defined with class JLabel—a subclass of JComponent. A label displays a single line of read-only text, an image or both text and an image. Programs rarely change a label’s contents after creating it. The application of Figure 12.4 demonstrates several JLabel features.

```java
// Fig. 12.4: LabelTest.java
// Demonstrating the JLabel class.

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

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

public class LabelTest extends JFrame {
    private JLabel label1, label2, label3;

    // set up GUI
    public LabelTest() {
        super( "Testing JLabel" );
    }
}
```

Fig. 12.4 Demonstrating class JLabel (part 1 of 2).
// get content pane and set its layout
Container container = getContentPane();
container.setLayout( new FlowLayout() );

// JLabel constructor with a string argument
label1 = new JLabel( "Label with text" );
label1.setToolTipText( "This is label1" );
container.add( label1 );

// JLabel constructor with string, Icon and
// alignment arguments
Icon bug = new ImageIcon( "bug1.gif" );
label2 = new JLabel( "Label with text and icon",
bug, SwingConstants.LEFT );
label2.setToolTipText( "This is label2" );
container.add( label2 );

// JLabel constructor no arguments
label3 = new JLabel();
label3.setText( "Label with icon and text at bottom" );
label3.setIcon( bug );
label3.setHorizontalTextPosition( SwingConstants.CENTER );
label3.setVerticalTextPosition( SwingConstants.BOTTOM );
label3.setToolTipText( "This is label3" );
container.add( label3 );

setSize( 275, 170 );
setVisible( true );

// execute application
public static void main( String args[] )
{
      LabelTest application = new LabelTest();
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

}  // end class LabelTest

Fig. 12.4 Demonstrating class JLabel (part 2 of 2).
Good Programming Practice 12.4  
Study the methods of class `javax.swing.JLabel` in the Java 2 SDK on-line documentation to learn the complete capabilities of the class before using it.

The program declares three `JLabel` objects at line 12. The `JLabel` objects are instantiated in the `LabelTest` constructor (line 15–47). Line 24 creates a `JLabel` object with the text "Label with text". The label displays this text when the label appears on the screen (i.e., when the window is displayed in this program).

Line 25 uses method `setToolTipText` (inherited into class `JLabel` from class `JComponent`) to specify the tool tip (see the second screen capture in Fig. 12.4) that is displayed automatically when the user positions the mouse cursor over the label in the GUI. When you execute this program, try positioning the mouse over each label to see its tool tip. Line 26 adds `label1` to the content pane.

Look-and-Feel Observation 12.3  
Use tool tips (set with `JComponent` method `setToolTipText`) to add descriptive text to your GUI components. This text helps the user determine the GUI component’s purpose in the user interface.

Several Swing components can display images by specifying an `Icon` as an argument to their constructor or by using a method that is normally called `setIcon`. An `Icon` is an object of any class that implements interface `Icon` (package `javax.swing`). One such class is `ImageIcon` (package `javax.swing`), which supports several image formats, including Graphics Interchange Format (GIF), Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG). File names for each of these types typically end with `.gif`, `.png` or `.jpg` (or `.jpeg`), respectively. We discuss images in more detail in Chapter 18, Multimedia. Line 30 defines an `ImageIcon` object. The file `bug1.gif` contains the image to load and store in the `ImageIcon` object. This file is assumed to be in the same directory as the program (we will discuss locating the file elsewhere in Chapter 18). The `ImageIcon` object is assigned to `Icon` reference `bug`. Remember, class `ImageIcon` implements interface `Icon`, therefore an `ImageIcon` is an `Icon`.

Class `JLabel` supports the display of `Icons`. Lines 31–32 use another `JLabel` constructor to create a label that displays the text "Label with text and icon" and the `Icon` to which `bug` refers and is left justified or left aligned (i.e., the icon and text are at the left side of the label’s area on the screen). Interface `SwingConstants` (package `javax.swing`) defines a set of common integer constants (such as `SwingConstants.LEFT`) that are used with many Swing components. By default, the text appears to the right of the image when a label contains both text and an image. The horizontal and vertical alignments of a label can be set with methods `setHorizontalAlignment` and `setVerticalAlignment`, respectively. Line 33 specifies the tool tip text for `label2`. Line 34 adds `label2` to the content pane.

Common Programming Error 12.1  
If you do not explicitly add a GUI component to a container, the GUI component will not be displayed when the container appears on the screen.

Common Programming Error 12.2  
Adding to a container a component that has not been instantiated throws a `NullPointerException`.
Class **JLabel** provides many methods to configure a label after it has been instantiated. Line 37 creates a **JLabel** and invokes the no-argument (default constructor). Such a label has no text or **Icon**. Line 38 uses **JLabel** method **setText** to set the text displayed on the label. A corresponding method **getText** retrieves the current text displayed on a label. Line 39 uses **JLabel** method **setIcon** to set the **Icon** displayed on the label. A corresponding method **getIcon** retrieves the current **Icon** displayed on a label. Lines 40–41 use **JLabel** methods **setHorizontalTextPosition** and **setVerticalTextPosition** to specify the text position in the label. In this case, the text will be centered horizontally and will appear at the bottom of the label. Thus, the **Icon** will appear above the text. Line 42 sets the tool tip text for the **label3**. Line 43 adds **label3** to the content pane.

### 12.4 Event-Handling Model

In the preceding section, we did not discuss event handling because there are no specific events for **JLabel** objects. GUIs are **event driven** (i.e., they generate **events** when the user of the program interacts with the GUI). Some common interactions are moving the mouse, clicking the mouse, clicking a button, typing in a text field, selecting an item from a menu, closing a window, etc. When a user interaction occurs, an event is sent to the program. GUI event information is stored in an object of a class that extends **AWTEvent**. Figure 12.5 illustrates a hierarchy containing many of the event classes we use from package **java.awt.event**. Many of these event classes are discussed throughout this chapter and Chapter 13. The event types from package **java.awt.event** are used with both AWT and Swing components. Additional event types have also been added that are specific to several types of Swing components. New Swing-component event types are defined in package **javax.swing.event**.

![Fig. 12.5 Some event classes of package java.awt.event.](image)
There are three parts to the event-handling mechanism—the event source, the event object and the event listener. The event source is the particular GUI component with which the user interacts. The event object encapsulates information about the event that occurred. This information includes a reference to the event source and any event-specific information that may be required by the event listener to handle the event. The event listener is an object that is notified by the event source when an event occurs. The event listener receives an event object when it is notified of the event, then uses the object to respond to the event. The event source is required to provide methods that enable listeners to be registered and unregistered. The event source also is required to maintain a list of its registered listeners and be able to notify its listeners when an event occurs.

The programmer must perform two key tasks to process a graphical user interface event in a program—register an event listener for the GUI component that is expected to generate the event, and implement an event handling method (or set of event-handling methods). Commonly, event-handling methods are called event handlers. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from package java.awt.event and package javax.swing.event.

Many of the event-listener types are common to both Swing and AWT components. Such types are defined in package java.awt.event, and many of these are shown in Fig. 12.6. Additional event-listener types that are specific to Swing components are defined in package javax.swing.event.
An event listener object “listens” for specific types of events generated by event sources (normally GUI components) in a program. An event handler is a method that is called in response to a particular type of event. Each event-listener interface specifies one or more event-handling methods that must be defined in the class that implements the event-listener interface. Remember that interfaces define abstract methods. Any class that implements an interface must define all the methods of that interface; otherwise, the class is an abstract class and cannot be used to create objects. The use of event listeners in event handling is known as the delegation event model—the processing of an event is delegated to a particular object (the listener) in the program.

When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener’s appropriate event handling method. For example, when the user presses the Enter key in a JTextField, the registered listener’s actionPerformed method is called. How did the event handler get registered? How does the GUI component know to call actionPerformed as opposed to some other event handling method? We answer these questions and diagram the interaction as part of the next example.

12.5 JTextField and JPasswordField

JTextField and JPasswordField (package javax.swing) are single-line areas in which text can be entered by the user from the keyboard or text can simply be displayed. A JPasswordField shows that a character was typed as the user enters characters, but hides the characters, assuming that they represent a password that should remain known only to the user. When the user types data into a JTextField or JPasswordField and presses the Enter key, an action event occurs. If the program registers an event listener, the listener processes the event and can use the data in the JTextField or JPasswordField at the time of the event in the program. Class JTextField extends class JTextComponent (package javax.swing.text), which provides many features common to Swing’s text-based components. Class JPasswordField extends JTextField and adds several methods that are specific to processing passwords.

Common Programming Error 12.3

Using a lowercase f in the class names JTextField or JPasswordField is a syntax error.

The application of Fig. 12.7 uses classes JTextField and JPasswordField to create and manipulate four fields. When the user presses Enter in the currently active field (the currently active component “has the focus”), a message dialog box containing the text in the field is displayed. When an event occurs in the JPasswordField, the password is revealed.

```java
1 // Fig. 12.7: TextFieldTest.java
2 // Demonstrating the JTextField class.
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
```

Fig. 12.7 Demonstrating JTextField and JPasswordField (part 1 of 4).
// Java extension packages
import javax.swing.*;

public class TextFieldTest extends JFrame {
    private JTextField textField1, textField2, textField3;
    private JPasswordField passwordField;

    // set up GUI
    public TextFieldTest()
    {
        super( "Testing JTextField and JPasswordField" );
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // construct textfield with default sizing
        textField1 = new JTextField( 10 );
        container.add( textField1 );

        // construct textfield with default text
        textField2 = new JTextField( "Enter text here" );
        container.add( textField2 );

        // construct textfield with default text and
        // 20 visible elements and no event handler
        textField3 = new JTextField( "Uneditable text field", 20 );
        textField3.setEditable( false );
        container.add( textField3 );

        // construct textfield with default text
        passwordField = new JPasswordField( "Hidden text" );
        container.add( passwordField );

        // register event handlers
        TextFieldHandler handler = new TextFieldHandler();
        textField1.addActionListener( handler );
        textField2.addActionListener( handler );
        textField3.addActionListener( handler );
        passwordField.addActionListener( handler );

        setSize( 325, 100 );
        setVisible( true );
    }

    // execute application
    public static void main( String args[] )
    {
        TextFieldTest application = new TextFieldTest();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}

Fig. 12.7 Demonstrating JTextField and JPasswordField (part 2 of 4).
// private inner class for event handling
private class TextFieldHandler implements ActionListener {

    // process text field events
    public void actionPerformed( ActionEvent event ) {
        String string = "";

        // user pressed Enter in JTextField textField1
        if ( event.getSource() == textField1 )
            string = "textField1: " + event.getActionCommand();

        // user pressed Enter in JTextField textField2
        else if ( event.getSource() == textField2 )
            string = "textField2: " + event.getActionCommand();

        // user pressed Enter in JTextField textField3
        else if ( event.getSource() == textField3 )
            string = "textField3: " + event.getActionCommand();

        // user pressed Enter in JTextField passwordField
        else if ( event.getSource() == passwordField ) {
            JPasswordField pwd = ( JPasswordField ) event.getSource();
            string = "passwordField: " +
            new String( passwordField.getPassword() );
        }

        JOptionPane.showMessageDialog( null, string );
    }
}

} // end class TextFieldTest
Lines 12–13 declare three references for `JTextField` (`textField1`, `textField2` and `textField3`) and a `JPasswordField` (`passwordField`). Each of these is instantiated in the constructor (line 16–50). Line 24 defines `JTextField textField1` with 10 columns of text. The width of the text field will be the width in pixels of the average character in the text field’s current font multiplied by 10. Line 25 adds `textField1` to the content pane.

Line 28 defines `textField2` with the initial text "Enter text here" to display in the text field. The width of the text field is determined by the text. Line 29 adds `textField2` to the content pane.

Line 33 defines `textField3` and call the `JTextField` constructor with two arguments—the default text "Uneditable text field" to display in the text field and the number of columns (20). The width of the text field is determined by the number of columns specified. Line 34 uses method `setEditable` (inherited into `JTextField` from class `JTextComponent`) to indicate that the user cannot modify the text in the text field. Line 35 adds `textField3` to the content pane.

Line 38 defines `JPasswordField passwordField` with the text "Hidden text" to display in the text field. The width of the text field is determined by the text. Notice that the text is displayed as a string of asterisks when the program executes. Line 39 adds `passwordField` to the content pane.

For the event-handling in this example, we defined inner class `TextFieldHandler` (lines 62–92), which implements interface `ActionListener` (class `TextFieldHandler` is discussed shortly). Thus, every instance of class `TextFieldHandler` is an `ActionListener`. Line 42 defines an instance of class `TextFieldHandler` and assigns it to reference `handler`. This one instance will be used as the event-listener object for the `JTextFields` and the `JPasswordField` in this example.

Lines 43–46 are the event registration statements that specify the event listener object for each of the three `JTextFields` and for the `JPasswordField`. After these statements execute, the object to which `handler` refers is listening for events (i.e., it will be notified when an event occurs) on these four objects. The program calls `JTextField` method `addActionListener` to register the event for each component. Method `addActionListener` receives as its argument an `ActionListener` object. Thus, any object of a class that implements interface `ActionListener` (i.e., any object that is an `Action-`
Listener) can be supplied as an argument to this method. The object to which \texttt{handler}
refers is an \texttt{ActionListener} because its class implements interface \texttt{Action-
Listener}. Now, when the user presses \texttt{Enter} in any of these four fields, method \texttt{action-
Performed} (line 65–90) in class \texttt{TextFieldHandler} is called to handle the event.

Software Engineering Observation 12.2

\textit{The event listener for an event must implement the appropriate event-listener interface.}

Method \texttt{actionPerformed} uses its \texttt{ActionEvent} argument’s method \texttt{getSource} to determine the GUI component with which the user interacted and creates a \texttt{String} to display in a message dialog box. \texttt{ActionEvent} method \texttt{getActionCom-
mand} returns the text in the \texttt{JTextField} that generated the event. If the user interacted
with the \texttt{JPasswordField}, lines 83–84 cast the \texttt{Component} reference returned by
\texttt{event.getSource()} to a \texttt{JPasswordField} reference so that lines 85–86 can use
\texttt{JPasswordField} method \texttt{getPassword} to obtain the password and create the
\texttt{String} to display. Method \texttt{getPassword} returns the password as an array of type
\texttt{char} that is used as an argument to a \texttt{String} constructor to create a \texttt{String}. Line 89
displays a message box indicating the GUI component reference name and the text the user
typed in the field.

Note that even an uneditable \texttt{JTextField} can generate an event. Simply click the
text field, then press \texttt{Enter}. Also note that the actual text of the password is displayed when
you press \texttt{Enter} in the \texttt{JPasswordField} (of course, you would normally not do this!).

Common Programming Error 12.4

\textit{Forgetting to register an event handler object for a particular GUI component’s event type
results in no events being handled for that component for that event type.}

Using a separate class to define an event listener is a common programming practice
for separating the GUI interface from the implementation of its event handler. For the
remainder of this chapter and Chapter 13, many programs use separate event-listener
classes to process GUI events.

12.5.1 How Event Handling Works

Let us illustrate how the event-handling mechanism works using \texttt{textField1} from the
preceding example. We have two remaining open questions from Section 12.4:

1. How did the event handler get registered?
2. How does the GUI component know to call \texttt{actionPerformed} as opposed to
   some other event handling method?

The first question is answered by the event registration performed in lines 43–46 of the pro-
gram. Figure 12.8 diagrams \texttt{JTextField} reference \texttt{textField1}, the \texttt{JTextField}
object to which it refers and the listener object that is registered to handle the \texttt{JText-
Field}’s event.

Every \texttt{JComponent} has an object of class \texttt{EventListenerList} (package
\texttt{javax.swing.event}) called \texttt{listenerList} as an instance variable. All registered
listeners are stored in the \texttt{listenerList} (diagramed as an array in Figure 12.8). When
the statement
textField1.addActionListener( handler );

executes in Fig. 12.7, a new entry is placed in the listenerList for JTextField textField1, indicating both the reference to the listener object and the type of listener (in this case ActionListener).

The type is important in answering the second question—how does the GUI component know to call actionPerformed rather than another event handling method? Every JComponent actually supports several different event types, including mouse events, key events and others. When an event occurs, the event is dispatched only to the event listeners of the appropriate type. The dispatching of an event is simply calling the event handling method for each registered listener for that event type.

Each event type has a corresponding event-listener interface. For example, ActionEvents are handled by ActionListener, MouseEvents are handled by MouseListeners (and MouseMotionListeners as we will see) and KeyEvents are handled by KeyListeners. When an event is generated by a user interaction with a component, the component is handed a unique event ID specifying the event type that occurred. The GUI component uses the event ID to decide the type of listener to which the event should be dispatched and the method to call. In the case of an ActionEvent, the event is dispatched to every registered ActionListener’s actionPerformed method (the only method in interface ActionListener). In the case of a MouseEvent, the event is dispatched to every registered MouseListener (or MouseMotionListener, depending on the event that occurs). The event ID of the MouseEvent determines which of the seven different mouse event handling methods are called. All of this decision logic is handled for you by the GUI components. We discuss other event types and event-listener interfaces as they are needed with each new component we cover.
12.6 JButton

A button is a component the user clicks to trigger a specific action. A Java program can use several types of buttons, including command buttons, check boxes, toggle buttons and radio buttons. Figure 12.9 shows the inheritance hierarchy of the Swing buttons we cover in this chapter. As you can see in the diagram, all the button types are subclasses of AbstractButton (package javax.swing), which defines many of the features that are common to Swing buttons. In this section, we concentrate on buttons that are typically used to initiate a command. Other button types are covered in the next several sections.

A command button generates an ActionEvent when the user clicks the button with the mouse. Command buttons are created with class JButton, which inherits from class AbstractButton. The text on the face of a JButton is called a button label. A GUI can have many JButtons, but each button label typically should be unique.

Look-and-Feel Observation 12.4

Having more than one JButton with the same label makes the JButtons ambiguous to the user. Be sure to provide a unique label for each button.

The application of Fig. 12.10 creates two JButtons and demonstrates that JButtons (like JLabels) support the display of Icons. Event handling for the buttons is performed by a single instance of inner class ButtonHandler (defined at lines 53–62).
// set up GUI
public ButtonTest()
{
    super( "Testing Buttons" );

    // get content pane and set its layout
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    // create buttons
    plainButton = new JButton( "Plain Button" );
    container.add( plainButton );

    Icon bug1 = new ImageIcon( "bug1.gif" );
    Icon bug2 = new ImageIcon( "bug2.gif" );
    fancyButton = new JButton( "Fancy Button", bug1 );
    fancyButton.setRolloverIcon( bug2 );
    container.add( fancyButton );

    // create an instance of inner class ButtonHandler
    // to use for button event handling
    ButtonHandler handler = new ButtonHandler();
    fancyButton.addActionListener( handler );
    plainButton.addActionListener( handler );

    setSize( 275, 100 );
    setVisible( true );
}

// execute application
public static void main( String args[] )
{
    ButtonTest application = new ButtonTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

// inner class for button event handling
private class ButtonHandler implements ActionListener {

    // handle button event
    public void actionPerformed( ActionEvent event )
    {
        JOptionPane.showMessageDialog( null, 
            "You pressed: " + event.getActionCommand() );
    }
}

Fig. 12.10 Demonstrating command buttons and action events (part 2 of 3).
Line 12 declares two references to instances of class `JButton`—`plainButton` and `fancyButton`—that are instantiated in the constructor.

Line 24 creates `plainButton` with the button label "Plain Button". Line 25 adds the button to the content pane.

A `JButton` can display `Icons`. To provide the user with an extra level of visual interactivity with the GUI, a `JButton` can also have a `rollover Icon`—an `Icon` that is displayed when the mouse is positioned over the button. The icon on the button changes as the mouse moves in and out of the button’s area on the screen. Lines 27–28 create two `ImageIcon` objects that represent the default `Icon` and rollover `Icon` for the `JButton` created at line 29. Both statements assume the image files are stored in the same directory as the program (this is commonly the case for applications that use images).

Line 29 creates `fancyButton` with default text "Fancy Button" and the `Icon` `bug1`. By default, the text is displayed to the right of the icon. Line 30 uses method `setRolloverIcon` (inherited from class `AbstractButton` into class `JButton`) to specify the image displayed on the button when the user positions the mouse over the button. Line 31 adds the button to the content pane.

**Look-and-Feel Observation 12.5**

Using rollover icons for `JButtons` provides the user with visual feedback indicating that if they click the mouse, the button’s action will occur.

`JButtons` (like `JTextFields`) generate `ActionEvents`. As mentioned previously, an `ActionEvent` can be processed by any `ActionListener` object. Lines 35–37 register an `ActionListener` object for each `JButton`. Inner class `ButtonHandler` (lines 53–62) defines `actionPerformed` to display a message dialog box con-

Fig. 12.10 Demonstrating command buttons and action events (part 3 of 3).
taining the label for the button that was pressed by the user. *ActionEvent* method
*getActionCommand* returns the label on the button that generated the event.

### 12.7 JCheckBox and JRadioButton

The Swing GUI components contain three types of *state buttons*—`JToggleButton`, `JCheckBox` and `JRadioButton`—that have on/off or true/false values. JToggle-
Buttons are frequently used with *toolbars* (sets of small buttons typically located on a bar
across the top of a window) and are covered in Chapter 13. Classes `JCheckBox` and `JRa-
dioButton` are subclasses of `JToggleButton`. A `JRadioButton` is different from a
`JCheckBox` in that there are normally several `JRadioButtons` that are grouped to-
gether and only one of the `JRadioButtons` in the group can be selected (true) at any
time. We first discuss class `JCheckBox`.

#### Look-and-Feel Observation 12.6

Because class `AbstractButton` supports displaying text and images on a button, all sub-
classes of `AbstractButton` also support displaying text and images.

The application of Fig. 12.11 uses two `JCheckBox` objects to change the font style
of the text displayed in a `JTextField`. One `JCheckBox` applies a bold style when
selected and the other applies an italic style when selected. If both are selected, the style of
the font is bold and italic. When the program initially executes, neither `JCheckBox` is
checked (**true**).

```java
// Fig. 12.11: CheckBoxTest.java
// Creating Checkbox buttons.
public class CheckBoxTest extends JFrame {
  private JTextField field;
  private JCheckBox bold, italic;

  public CheckBoxTest() {
    super( "JCheckBox Test" );

    // get content pane and set its layout
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    // set up JTextField and set its font
    field = new JTextField( "Watch the font style change", 20 );
```

---

**Fig. 12.11** Program that creates two `JCheckBox` buttons (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
field.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
container.add( field );

// create checkbox objects
bold = new JCheckBox( "Bold" );
container.add( bold );
italic = new JCheckBox( "Italic" );
container.add( italic );

// register listeners for JCheckBoxes
CheckBoxHandler handler = new CheckBoxHandler();
bold.addItemListener( handler );
italic.addItemListener( handler );

setSize( 275, 100 );
setVisible( true );

// execute application
public static void main( String args[] )
{
  CheckBoxTest application = new CheckBoxTest();
  application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

// private inner class for ItemListener event handling
private class CheckBoxHandler implements ItemListener {
  private int valBold = Font.PLAIN;
  private int valItalic = Font.PLAIN;

  // respond to checkbox events
  public void itemStateChanged( ItemEvent event )
  {
    // process bold checkbox events
    if ( event.getSource() == bold )
      if ( event.getStateChange() == ItemEvent.SELECTED )
        valBold = Font.BOLD;
      else
        valBold = Font.PLAIN;

    // process italic checkbox events
    if ( event.getSource() == italic )
      if ( event.getStateChange() == ItemEvent.SELECTED )
        valItalic = Font.ITALIC;
      else
        valItalic = Font.PLAIN;
  }
}
After the `JTextField` is created and initialized, line 27 sets the font of the `JTextField` to `Serif`, `PLAIN` style and 14-point size. Next, the constructor creates two `JCheckBox` objects with lines 31 and 34. The `String` passed to the constructor is the `check box label` that appears to the right of the `JCheckBox` by default.

When the user clicks a `JCheckBox`, an `ItemEvent` occurs that can be handled by an `ItemListener` (any object of a class that implements interface `ItemListener`). An `ItemListener` must define method `itemStateChanged`. In this example, the event handling is performed by an instance of inner class `CheckBoxHandler` (lines 56–84). Lines 38–40 create an instance of class `CheckBoxHandler` and register it with method `addItemListener` as the `ItemListener` for both the `bold` and `italic` `JCheckBoxes`.

Method `itemStateChanged` (lines 61–82) is called when the user clicks either the `bold` or the `italic` checkbox. The method uses `event.getSource()` to determine which `JCheckBox` was clicked. If it was `JCheckBox bold`, the `if/else` structure at lines 66–69 uses `ItemEvent` method `getStateChange` to determine the state of the button (`ItemEvent.SELECTED` or `ItemEvent.DESELECTED`). If the state is selected, integer `valBold` is assigned `Font.BOLD`; otherwise, `valBold` is assigned `Font.PLAIN`. A similar `if/else` structure is executed if `JCheckBox italic` is clicked. If the `italic` state is selected, integer `valItalic` is assigned `Font.ITALIC`; otherwise, `valItalic` is assigned `Font.PLAIN`. The sum of `valBold` and `valItalic` is used at lines 80–81 as the style of the new font for the `JTextField`.

Radio buttons (defined with class `JRadioButton`) are similar to check boxes in that they have two states—`selected` and `not selected` (also called `deselected`). However, radio buttons normally appear as a `group` in which only one radio button can be selected at a time. Selecting a different radio button in the group automatically forces all other radio buttons in the group to be deselected. Radio buttons are used to represent a set of `mutually exclusive`
options (i.e., multiple options in the group would not be selected at the same time). The logical relationship between radio buttons is maintained by a `ButtonGroup` object (package `javax.swing`). The `ButtonGroup` object itself is not a GUI component. Therefore, a `ButtonGroup` object is not displayed in a user interface. Rather, the individual `JRadioButton` objects from the group are displayed in the GUI.

**Common Programming Error 12.5**

Adding a `ButtonGroup` object (or an object of any other class that does not derive from `Component`) to a container is a syntax error.

The application of Fig. 12.12 is similar to the preceding program. The user can alter the font style of a `JTextField`'s text. The program uses radio buttons that permit only a single font style in the group to be selected at a time.

```java
// Fig. 12.12: RadioButtonTest.java
// Creating radio buttons using ButtonGroup and JRadioButton.

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

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

public class RadioButtonTest extends JFrame {
    private JTextField field;
    private Font plainFont, boldFont, italicFont, boldItalicFont;
    private JRadioButton plainButton, boldButton, italicButton,
    boldItalicButton;
    private ButtonGroup radioGroup;

    // create GUI and fonts
    public RadioButtonTest()
    {
        super( "RadioButton Test" );

        // get content pane and set its layout
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        // set up JTextField
        field =
            new JTextField( "Watch the font style change", 25 );
        container.add( field );

        // create radio buttons
        plainButton = new JRadioButton( "Plain", true );
        container.add( plainButton );
        boldButton = new JRadioButton( "Bold", false );
        container.add( boldButton );
    }
}
```

Fig. 12.12 Creating and manipulating radio button (part 1 of 3).
italicButton = new JRadioButton( "Italic", false );
container.add( italicButton );

boldItalicButton = new JRadioButton( "Bold/Italic", false );
container.add( boldItalicButton );

// register events for JRadioButtons
RadioButtonHandler handler = new RadioButtonHandler();
plainButton.addItemListener( handler );
boldButton.addItemListener( handler );
italicButton.addItemListener( handler );
boldItalicButton.addItemListener( handler );

// create logical relationship between JRadioButtons
radioGroup = new ButtonGroup();
radioGroup.add( plainButton );
radioGroup.add( boldButton );
radioGroup.add( italicButton );
radioGroup.add( boldItalicButton );

// create font objects
plainFont = new Font( "Serif", Font.PLAIN, 14 );
boldFont = new Font( "Serif", Font.BOLD, 14 );
italicFont = new Font( "Serif", Font.ITALIC, 14 );
boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
field.setFont( plainFont );

setSize( 300, 100 );
setVisible( true );

// execute application
public static void main( String args[] )
{
    RadioButtonTest application = new RadioButtonTest();

    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

// private inner class to handle radio button events
private class RadioButtonHandler implements ItemListener {
    // handle radio button events
    public void itemStateChanged( ItemEvent event )
    {
        // user clicked plainButton
        if ( event.getSource() == plainButton )
            field.setFont( plainFont );

Lines 33–44 in the constructor define each JRadioButton object and add it to the application window’s content pane. Each JRadioButton is initialized with a constructor call like line 33. This constructor supplies the label that appears to the right of the JRadioButton by default and the initial state of the JRadioButton. A true second argument indicates that the JRadioButton should appear selected when it is displayed.

JRadioButtons, like JCheckBoxes, generate ItemEvents when they are clicked. Lines 47–51 create an instance of inner class RadioButtonHandler (defined at lines 82–104) and register it to handle the ItemEvent generated when the user clicks any one of the JRadioButtons.

Line 54 instantiates a ButtonGroup object and assigns it to reference radioGroup. This object is the “glue” that binds the four JRadioButton objects together to form the logical relationship that allows only one of the four buttons to be selected at a time. Lines 55–58 use ButtonGroup method add to associate each of the JRadioButtons with radioGroup. If more than one selected JRadioButton object is added to the group, the first selected JRadioButton added will be selected when the GUI is displayed.

Class RadioButtonHandler (line 82–104) implements interface ItemListener so it can handle item events generated by the JRadioButtons. Each JRadioButton in the program has an instance of this class (handler) registered as its ItemListener. When the user clicks a JRadioButton, radioGroup turns off the previously selected JRadioButton and method itemStateChanged (line 85–102)
executes. The method determines which \texttt{JRadioButton} was clicked using method \texttt{getSource} (inherited indirectly from \texttt{EventObject} into \texttt{ItemEvent}), then sets the font in the \texttt{JTextField} to one of the \texttt{Font} objects created in the constructor.

12.8 \texttt{JComboBox}

A combo box (sometimes called a drop-down list) provides a list of items from which the user can make a selection. Combo boxes are implemented with class \texttt{JComboBox}, which inherits from class \texttt{JComponent}. \texttt{JComboBox}es generate \texttt{ItemEvent}s like \texttt{JCheckBox}es and \texttt{JRadioButton}s.

The application of Fig. 12.13 uses a \texttt{JComboBox} to provide a list of four image file names. When an image file name is selected, the corresponding image is displayed as an \texttt{Icon} on a \texttt{JLabel}. The screen captures for this program show the \texttt{JComboBox} list after the selection was made to illustrate which image file name was selected.

Lines 17–19 declare and initialize array icons with four new \texttt{ImageIcon} objects. \texttt{String} array \texttt{names} (defined on lines 15–16) contains the names of the four image files that are stored in the same directory as the application.

Line 31 creates a \texttt{JComboBox} object, using the \texttt{Strings} in array \texttt{names} as the elements in the list. A numeric \texttt{index} keeps track of the ordering of items in the \texttt{JComboBox}. The first item is added at index 0; the next item is added at index 1, and so forth. The first item added to a \texttt{JComboBox} appears as the currently selected item when the \texttt{JComboBox} is displayed. Other items are selected by clicking the \texttt{JComboBox}. When clicked, the \texttt{JComboBox} expands into a list from which the user can make a selection.

Line 32 uses \texttt{JComboBox} method \texttt{setMaximumRowCount} to set the maximum number of elements that are displayed when the user clicks the \texttt{JComboBox}. If there are more items in the \texttt{JComboBox} than the maximum number of elements that are displayed, the \texttt{JComboBox} automatically provides a scrollbar (see the first screen capture) that allows the user to view all the elements in the list. The user can click the scroll arrows at the top and bottom of the scrollbar to move up and down through the list one element at a time, or the user can drag the scroll box in the middle of the scrollbar up and down to move through the list. To drag the scroll box, hold the mouse button down with the mouse cursor on the scroll box and move the mouse.

```java
// Fig. 12.13: ComboBoxTest.java
// Using a JComboBox to select an image to display.

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

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

public class ComboBoxTest extends JFrame {
    private JComboBox imagesComboBox;
    private JLabel label;
```

\textbf{Fig. 12.13} Program that uses a \texttt{JComboBox} to select an icon (part 1 of 3).
private String names[] =
{ "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };

private Icon icons[] = {
    new ImageIcon( names[ 0 ] ),
    new ImageIcon( names[ 1 ] ),
    new ImageIcon( names[ 2 ] ),
    new ImageIcon( names[ 3 ] )
};

// set up GUI
public ComboBoxTest()
{
    super( "Testing JComboBox" );

    // get content pane and set its layout
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    // set up JComboBox and register its event handler
    imagesComboBox = new JComboBox( names );
    imagesComboBox.setMaximumRowCount( 3 );

    imagesComboBox.addItemListener(

        // anonymous inner class to handle JComboBox events
        new ItemListener() {

            // handle JComboBox event
            public void itemStateChanged( ItemEvent event )
            {

                // determine whether check box selected
                if ( event.getStateChange() == ItemEvent.SELECTED )
                    label.setIcon( icons[ imagesComboBox.getSelectedIndex() ] );
            }

        }); // end anonymous inner class

    container.add( imagesComboBox );

    // set up JLabel to display ImageIcons
    label = new JLabel( icons[ 0 ] );
    container.add( label );

    setSize( 350, 100 );
    setVisible( true );
}

// execute application
public static void main( String args[] )
{
    ComboBoxTest application = new ComboBoxTest();
}

Fig. 12.13 Program that uses a JComboBox to select an icon (part 2 of 3).
Look-and-Feel Observation 12.7

Set the maximum row count for a JComboBox to a number of rows that prevents the list from expanding outside the bounds of the window or applet in which it is used. This will ensure that the list displays correctly when it is expanded by the user.

Lines 34–50 register an instance of an anonymous inner class that implements ItemListener as the listener for JComboBox images. When the user makes a selection from images, method itemStateChanged (line 40–46) sets the Icon for label. The Icon is selected from array icons by determining the index number of the selected item in the JComboBox with method getSelectedIndex in line 45. Note that line 43 changes the icon only for a selected item. The reason for the if structure here is that for each item that is selected from a JComboBox, another item is deselected. Thus, two events occur for each item selected. We wish to display only the icon for the item the user just selected.

12.9 JList

A list displays a series of items from which the user may select one or more items. Lists are created with class JList, which inherits from class JComponent. Class JList supports single-selection lists (i.e., lists that allow only one item to be selected at a time) and multiple-selection lists (lists that allow any number of items to be selected). In this section, we discuss single-selection lists.
The application of Fig. 12.14 creates a JList of 13 colors. When a color name is clicked in the JList, a ListSelectionEvent occurs and the application window content pane’s background color changes.

```java
public class ListTest extends JFrame {
    private JList colorList;
    private Container container;


    private Color colors[] = { Color.black, Color.blue, Color.cyan, Color.darkGray, Color.gray, Color.green, Color.lightGray, Color.magenta, Color.orange, Color.pink, Color.red, Color.white, Color.yellow };

    public ListTest() {
        super( "List Test" );

        // get content pane and set its layout
        container = getContentPane();
        container.setLayout( new FlowLayout() );

        // create a list with items in colorNames array
        colorList = new JList( colorNames );
        colorList.setVisibleRowCount( 5 );

        // do not allow multiple selections
        colorList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );

        // add a JScrollPane containing JList to content pane
        container.add( new JScrollPane( colorList ) );

        // set up event handler
        colorList.addListSelectionListener(
            new ListSelectionListener() {
                // anonymous inner class for list selection events
                new ListSelectionListener()
```
A `JList` object is instantiated at line 34 and assigned to reference `colorList` in the constructor. The argument to the `JList` constructor is the array of `Object`s (in this case `String`s) to display in the list. Line 35 uses `JList` method `setVisibleRowCount` to determine the number of items that are visible in the list.

Lines 38–39 use `JList` method `setSelectionMode` to specify the list selection mode. Class `ListSelectionModel` (package `javax.swing`) defines constants `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` and `MULTIPLE_INTERVAL_SELECTION` to specify a `JList`’s selection mode. A `SINGLE_SELECTION` list allows only one item to be selected at a time. A `SINGLE_INTERVAL_SELECTION` list is a multiple-selection list that allows several items in a contiguous range in the list to be selected. A `MULTIPLE_INTERVAL_SELECTION` list is a multiple-selection list that does not restrict the items that can be selected.

Unlike `JComboBox`, `JLists` do not provide a scrollbar if there are more items in the list than the number of visible rows. In this case, a `JScrollPane` object is used to provide...
the automatic scrolling capability for the JList. Line 42 adds a new instance of class JScrollPane to the content pane. The JScrollPane constructor receives as its argument the JComponent for which it will provide automatic scrolling functionality (in this case JList colorList). Notice in the screen captures that a scrollbar created by the JScrollPane appears at the right side of the JList. By default, the scrollbar appears only when the number of items in the JList exceeds the number of visible items.

Lines 45–59 use JList method addListSelectionListener to register an instance of an anonymous inner class that implements ListSelectionListener (defined in package javax.swing.event) as the listener for JList colorList. When the user makes a selection from colorList, method valueChanged (line 51–55) executes and sets the background color of the content pane with method setBackground (inherited from class Component into class Container). The color is selected from the array colors with the selected item’s index in the list that is returned by JList method getSelectedIndex (as with arrays, JList indexing is zero based).

### 12.10 Multiple-Selection Lists

A multiple-selection list enables the user to select many items from a JList. A SINGLE_INTERVAL_SELECTION list allows selection of a contiguous range of items in the list by clicking the first item, then holding the Shift key while clicking the last item to select in the range. A MULTIPLE_INTERVAL_SELECTION list allows continuous range selection as described for a SINGLE_INTERVAL_SELECTION list and allows miscellaneous items to be selected by holding the Ctrl key (sometimes called to Control key) while clicking each item to select. To deselect an item, hold the Ctrl key while clicking the item a second time.

The application of Fig. 12.15 uses multiple-selection lists to copy items from one JList to another. One list is a MULTIPLE_INTERVAL_SELECTION list and the other is a SINGLE_INTERVAL_SELECTION list. When you execute the program, try using the selection techniques described above to select items in both lists.

```java
12 // Fig. 12.15: MultipleSelection.java
13 // Copying items from one List to another.

14 // Java core packages
15 import java.awt.*;
16 import java.awt.event.*;
17
18 // Java extension packages
19 import javax.swing.*;
20
21 // Fig. 12.15: MultipleSelection.java
22 // Copying items from one List to another.
23
24 public class MultipleSelection extends JFrame {
25     private JList colorList, copyList;
26     private JButton copyButton;
27
28     private String colorNames[] = { "Black", "Blue", "Cyan",
29                                      "Dark Gray", "Gray", "Green", "Light Gray",
30                                      "Magenta", "Orange", "Pink", "Red", "White", "Yellow" };
31
32 }  // end class MultipleSelection
```

Fig. 12.15 Using a multiple-selection JList (part 1 of 3).
// set up GUI
public MultipleSelection()
{
    super( "Multiple Selection Lists" );

    // get content pane and set its layout
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    // set up JList colorList
    colorList = new JList( colorNames );
    colorList.setVisibleRowCount( 5 );
    colorList.setFixedCellHeight( 15 );
    colorList.setSelectionModel(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
    container.add( new JScrollPane( colorList ) );

    // create copy button and register its listener
    copyButton = new JButton( "Copy >>>" );

    copyButton.addActionListener(
        new ActionListener() {
            public void actionPerformed( ActionEvent event ) {
                // place selected values in copyList
                copyList.setListData(
                    colorList.getSelectedValues() );
            }
        }); // end anonymous inner class
    container.add( copyButton );

    // set up JList copyList
    copyList = new JList( );
    copyList.setVisibleRowCount( 5 );
    copyList.setFixedCellWidth( 100 );
    copyList.setFixedCellHeight( 15 );
    copyList.setSelectionMode(
        ListSelectionModel.SINGLE_INTERVAL_SELECTION );
    container.add( new JScrollPane( copyList ) );

    setSize( 300, 120 );
    setVisible( true );
}

Fig. 12.15 Using a multiple-selection JList (part 2 of 3).
Line 29 creates `JList colorList` and initializes it with the `Strings` in the array `colorNames`. Line 30 sets the number of visible rows in `colorList` to 5. Line 31 uses `JList` method `setFixedCellHeight` to specify the height in pixels of each item in the `JList`. We do this to ensure that the rows in both `JLists` in the example have the same height. Lines 32–33 specify that `colorList` is a `MULTIPLE_INTERVAL_SELECTION` list. Line 34 adds a new `JScrollPane` containing `colorList` to the content pane. Lines 59–65 perform similar tasks for `JList copyList`, which is defined as a `SINGLE_INTERVAL_SELECTION` list. Line 61 uses `JList` method `setFixedCellWidth` to set `copyList`'s width to 100 pixels.

A multiple-selection list does not have a specific event associated with making multiple selections. Normally, an event generated by another GUI component (known as an external event) specifies when the multiple selections in a `JList` should be processed. In this example, the user clicks `JButton copyButton` to trigger the event that copies the selected items in `colorList` to `copyList`.

When the user clicks `copyButton`, method `actionPerformed` (line 45–50) is called. Lines 48–49 use `JList` method `setListData` to set the items displayed in `copyList`. Line 49 calls `colorList`'s method `getSelectedValues`, which returns an array of `Objects` representing the selected items in `colorList`. In this example, the returned array is passed as the argument to `copyList`'s `setListData` method.

Many students ask how reference `copyList` can be used in line 48, when the program does not create the object to which it refers until Line 59. Remember that method `actionPerformed` at lines 45–50 does not execute until the user presses the `copyButton`, which cannot occur until after the call to the constructor completes. At that point in the program’s execution, line 59 already has initialized `copyList` with a new `JList` object.

### 12.11 Mouse Event Handling

This section presents the `MouseListener` and `MouseMotionListener` event-listener interfaces for handling `mouse events`. Mouse events can be trapped for any GUI com-
ponent that derives from `java.awt.Component`. The methods of interfaces `MouseListener` and `MouseMotionListener` are summarized in Figure 12.16.

Each of the mouse event handling methods takes a `MouseEvent` object as its argument. A `MouseEvent` object contains information about the mouse event that occurred, including the x- and y-coordinates of the location where the event occurred. The `MouseListener` and `MouseMotionListener` methods are called automatically when the mouse interacts with a `Component` if listener objects are registered for a particular `Component`. Method `mousePressed` is called when a mouse button is pressed with the mouse cursor over a component. Using methods and constants of class `InputEvent` (the superclass of `MouseEvent`), a program can determine which mouse button the user clicked. Method `mouseClicked` is called whenever a mouse button is released without moving the mouse after a `mousePressed` operation. Method `mouseReleased` is called whenever a mouse button is released. Method `mouseEntered` is called when the mouse cursor enters the physical boundaries of a `Component`. Method `mouseExited` is called when the mouse cursor leaves the physical boundaries of a `Component`. Method `mouseDragged` is called when the mouse button is pressed and held, and the mouse is moved (a process known as dragging). The `mouseDragged` event is preceded by a `mousePressed` event and followed by a `mouseReleased` event. Method `mouseMoved` is called when the mouse is moved with the mouse cursor over a component (and no mouse buttons pressed).

### MouseListener and MouseMotionListener interface methods

**Methods of interface MouseListener**

- **public void mousePressed(MouseEvent event)**
  Called when a mouse button is pressed with the mouse cursor on a component.

- **public void mouseClicked(MouseEvent event)**
  Called when a mouse button is pressed and released on a component without moving the mouse cursor.

- **public void mouseReleased(MouseEvent event)**
  Called when a mouse button is released after being pressed. This event is always preceded by a `mousePressed` event.

- **public void mouseEntered(MouseEvent event)**
  Called when the mouse cursor enters the bounds of a component.

- **public void mouseExited(MouseEvent event)**
  Called when the mouse cursor leaves the bounds of a component.

**Methods of interface MouseMotionListener**

- **public void mouseDragged(MouseEvent event)**
  Called when the mouse button is pressed with the mouse cursor on a component and the mouse is moved. This event is always preceded by a call to `mousePressed`.

- **public void mouseMoved(MouseEvent event)**
  Called when the mouse is moved with the mouse cursor on a component.
Look-and-Feel Observation 12.8

Method calls to `mouseDragged` are sent to the `MouseMotionListener` for the Component on which the drag operation started. Similarly, the `mouseReleased` method call is sent to the `MouseListener` for the Component on which the drag operation started.

The `MouseTracker` application (Fig. 12.17) demonstrates the `MouseListener` and `MouseMotionListener` methods. The application class implements both interfaces so it can listen for its own mouse events. Note that all seven methods from these two interfaces must be defined by the programmer when a class implements both interfaces. The message dialog box in the sample output windows appears when the user moves the mouse into the application window.

```java
// Fig. 12.17: MouseTracker.java
// Demonstrating mouse events.

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

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

public class MouseTracker extends JFrame
    implements MouseListener, MouseMotionListener {
    private JLabel statusBar;

    // set up GUI and register mouse event handlers
    public MouseTracker() {
        super( "Demonstrating Mouse Events" );

        statusBar = new JLabel();
        getContentPane().add( statusBar, BorderLayout.SOUTH );

        // application listens to its own mouse events
        addMouseListener( this );
        addMouseMotionListener( this );

        setSize( 275, 100 );
        setVisible( true );
    }

    // MouseListener event handlers

    // handle event when mouse released immediately after press
    public void mouseClicked( MouseEvent event ) {
        statusBar.setText( "Clicked at [" + event.getX() + ", " + event.getY() + "]" );
    }

    Fig. 12.17 Demonstrating mouse event handling (part 1 of 3).
```
// handle event when mouse pressed
public void mousePressed( MouseEvent event )
{
    statusBar.setText( "Pressed at [" + event.getX() + ", " + event.getY() + "]" );
}

// handle event when mouse released after dragging
public void mouseReleased( MouseEvent event )
{
    statusBar.setText( "Released at [" + event.getX() + ", " + event.getY() + "]" );
}

// handle event when mouse enters area
public void mouseEntered( MouseEvent event )
{
    JOptionPane.showMessageDialog( null, "Mouse in window" );
}

// handle event when mouse exits area
public void mouseExited( MouseEvent event )
{
    statusBar.setText( "Mouse outside window" );
}

// MouseMotionListener event handlers

// handle event when user drags mouse with button pressed
public void mouseDragged( MouseEvent event )
{
    statusBar.setText( "Dragged at [" + event.getX() + ", " + event.getY() + "]" );
}

// handle event when user moves mouse
public void mouseMoved( MouseEvent event )
{
    statusBar.setText( "Moved at [" + event.getX() + ", " + event.getY() + "]" );
}

// execute application
public static void main( String args[] )
{
    MouseTracker application = new MouseTracker();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

Fig. 12.17 Demonstrating mouse event handling (part 2 of 3).
Each mouse event results in a `String` displayed in `JLabel statusBar` at the bottom of the window.

Lines 21–22 in the constructor define `JLabel statusBar` and attach it to the content pane. Until now, each time we used the content pane, method `setLayout` was called to set the content pane’s layout manager to a `FlowLayout`. This allowed the content pane to display the GUI components we attached to it from left to right. If the GUI components do not fit on one line, the `FlowLayout` creates additional lines to continue displaying the GUI components. Actually, the default layout manager is a `BorderLayout` that divides the content pane’s area into five regions—north, south, east, west and center. Line 22 uses a new version of `Container` method `add` to attach `statusBar` to the region `BorderLayout.SOUTH`, which extends across the entire bottom of the content pane. We discuss `BorderLayout` and several other layout managers in detail later in this chapter.

Lines 25–26 in the constructor register the `MouseTracker` window object as the listener for its own mouse events. Methods `addMouseListener` and `addMouseMotionListener` are `Component` methods that can be used to register mouse event listeners for an object of any class that extends `Component`.

When the mouse enters or exits the application area, method `mouseEntered` (lines 56–59) and method `mouseExited` (lines 62–65) are called, respectively. Method `mouseExited` displays a message in `statusBar` indicating that the mouse is outside the application (see the first sample output window). Method `mouseEntered` displays a message dialog box indicating that the mouse entered the application window. [Note: Be sure to press `Enter` to dismiss the message dialog, rather than using the mouse. If you use the mouse to dismiss the dialog, when you move the mouse over the window again, `mouseEntered` redisplays the dialog. This will prevent you from trying the other mouse events.]
When any of the other five events occur, they display a message in \texttt{statusBar} that includes a \texttt{String} that represents the event that occurred and the coordinates where the mouse event occurred. The \(x\) and \(y\) coordinates of the mouse when the event occurred are obtained with \texttt{MouseEvent} methods \texttt{getX} and \texttt{getY}, respectively.

### 12.12 Adapter Classes

Many of the event-listener interfaces provide multiple methods; \texttt{MouseListener} and \texttt{MouseMotionListener} are examples. It is not always desirable to define every method in an event-listener interface. For example, a program may only need the \texttt{mouseClicked} handler from interface \texttt{MouseListener} or the \texttt{mouseDragged} handler from \texttt{MouseMotionListener}. In our windowed applications (subclasses of \texttt{JFrame}) terminating the application has been handled with \texttt{windowClosing} from interface \texttt{WindowListener}, which actually specifies seven window-event-handling methods. For many of the listener interfaces that contain multiple methods, package \texttt{java.awt.event} and package \texttt{javax.swing.event} provide event-listener adapter classes. An adapter class implements an interface and provides a default implementation (with an empty method body) of every method in the interface. The \texttt{java.awt.event} adapter classes are shown in Fig. 12.18 along with the interfaces they implement.

The programmer can extend the adapter class to inherit the default implementation of every method, then override the method(s) needed for event handling. The default implementation of each method in the adapter class has an empty body. This is exactly what we have been doing in each application example that extends \texttt{JFrame} and defines method \texttt{windowClosing} to handle the closing of the window and termination of the application.

#### Software Engineering Observation 12.3

When a class implements an interface, the class has an “is a” relationship with that interface. All direct and indirect subclasses of that class inherit this relationship. Thus, an object of a class that extends an event adapter class is an object of the corresponding event listener type (e.g., an object of a subclass of \texttt{MouseAdapter} is a \texttt{MouseListener}).

<table>
<thead>
<tr>
<th>Event adapter class</th>
<th>Implements interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComponentAdapter</td>
<td>ComponentListener</td>
</tr>
<tr>
<td>ContainerAdapter</td>
<td>ContainerListener</td>
</tr>
<tr>
<td>FocusAdapter</td>
<td>FocusListener</td>
</tr>
<tr>
<td>KeyAdapter</td>
<td>KeyListener</td>
</tr>
<tr>
<td>MouseAdapter</td>
<td>MouseListener</td>
</tr>
<tr>
<td>MouseMotionAdapter</td>
<td>MouseMotionListener</td>
</tr>
<tr>
<td>WindowAdapter</td>
<td>WindowListener</td>
</tr>
</tbody>
</table>

\textit{Fig. 12.18} Event adapter classes and the interfaces they implement.
The Painter application of Fig. 12.19 uses the mouseDragged event handler to create a simple drawing program. The user can draw pictures with the mouse by dragging the mouse on the background of the window. This example does not use method mouseMoved, so our MouseMotionListener is defined as a subclass of MouseMotionAdapter. This class already defines both mouseMoved and mouseDragged, so we can simply override mouseDragged to provide the drawing functionality.

```java
// Fig. 12.19: Painter.java
// Using class MouseMotionAdapter.

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

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

public class Painter extends JFrame {
    private int xValue = -10, yValue = -10;

    // set up GUI and register mouse event handler
    public Painter() {
        super( "A simple paint program" );

        // create a label and place it in SOUTH of BorderLayout
        getContentPane().add(
            new Label( "Drag the mouse to draw" ),
            BorderLayout.SOUTH );

        addMouseMotionListener(
            // anonymous inner class
            new MouseMotionAdapter() {

                // store drag coordinates and repaint
                public void mouseDragged( MouseEvent event )
                {
                    xValue = event.getX();
                    yValue = event.getY();
                    repaint();
                }

            } // end anonymous inner class
        ); // end call to addMouseMotionListener

        setSize( 300, 150 );
        setVisible( true );
    }
}
```

Fig. 12.19 Program that demonstrates adapter classes (part 1 of 2).
The instance variables `xValue` and `yValue` store the coordinates of the `mouseDragged` event. Initially, the coordinates are set outside the window area to prevent an oval from drawing on the background area in the first call to `paint` when the window is displayed. Lines 24–39 register a `MouseMotionListener` to listen for the window’s mouse motion events (remember that a call to a method that is not preceded by a reference and a dot operator is really preceded by "this.", indicating that the method is called for the current instance of the class at execution time). Lines 27–37 define an anonymous inner class that extends class `MouseMotionAdapter` (which implements `MouseMotionListener`). The anonymous inner class inherits a default implementation of both method `mouseMoved` and method `mouseDragged`. Thus, the anonymous inner class already satisfies the requirement that in all methods an interface must be implemented. However,
the default methods do nothing when they are called. So, we override method `mouseDragged` at lines 30–35 to capture the x- and y-coordinates of the mouse-dragged event and store them in instance variables `xValue` and `yValue`, then call `repaint` to initiate drawing the next oval on the background (performed by method `paint` at lines 47–53). Note that `paint` does not call the superclass version of `paint` inherited from `JFrame`. The superclass version normally clears the background of the window. Not calling the superclass version enables our program to keep all the ovals on the window at once. However, notice that if you cover the window with another window, only the last oval displayed still appears, because our program does not keep track of all the ovals displayed previously.

Lines 60–72 register a `WindowListener` to listen for the application window’s window events (such as closing the window). Lines 63–70 define an anonymous inner class that extends class `WindowAdapter` (which implements `WindowListener`). The anonymous inner class inherits a default implementation of seven different window-event-handler methods. Thus, the anonymous inner class already satisfies the requirement that in all methods an interface must be implemented. However, the default methods do nothing when they are called. So, we override method `windowClosing` at lines 65–68 to terminate the application when the user clicks the application window’s close box.

The `MouseDetails` application of Fig. 12.20 demonstrates how to determine the number of mouse clicks (i.e., the click count) and how to distinguish between the different mouse buttons. The event listener in this program is an object of inner class `MouseClickHandler` (lines 47–75) that extends `MouseAdapter` so we can define just the `mouseClicked` method we need in this example.

```java
// Fig. 12.20: MouseDetails.java
// Demonstrating mouse clicks and distinguishing between mouse buttons.

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

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

public class MouseDetails extends JFrame {
    private int xPos, yPos;

    // set title bar String, register mouse listener and size
    // and show window
    public MouseDetails() {
        super( "Mouse clicks and buttons" );
        addMouseListener( new MouseClickHandler() );
        setSize( 350, 150 );
        setVisible( true );
    }

    // Fig. 12.20 Distinguishing among left, center and right mouse-button clicks (part 1 of 3).
```
// draw String at location where mouse was clicked
public void paint( Graphics g )
{
    // call superclass's paint method
    super.paint( g );

    g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]", 
                  xPos, yPos );
}

// execute application
public static void main( String args[] )
{
    MouseDetails application = new MouseDetails();
    application.setDefaultCloseOperation( 
        JFrame.EXIT_ON_CLOSE );
}

// inner class to handle mouse events
private class MouseClickHandler extends MouseAdapter {

    // handle mouse click event and determine which mouse
    // button was pressed
    public void mouseClicked( MouseEvent event )
    {
        xPos = event.getX();
        yPos = event.getY();

        String title = 
            "Clicked " + event.getClickCount() + " time(s)";

        // right mouse button
        if ( event.isMetaDown() )
            title += " with right mouse button";

        // middle mouse button
        else if ( event.isAltDown() )
            title += " with center mouse button";

        // left mouse button
        else
            title += " with left mouse button";

        setTitle( title );  // set title bar of window
                          // set title bar of window
        repaint();
    }
} // end private inner class MouseClickHandler

// end class MouseDetails

Fig. 12.20  Distinguishing among left, center and right mouse-button clicks (part 2 of 3).
A user of a Java program may be on a system with a one-, two- or three-button mouse. Java provides a mechanism to distinguish among mouse buttons. Class `MouseEvent` inherits several methods from class `InputEvent` that can distinguish between mouse buttons on a multi-button mouse or can mimic a multi-button mouse with a combined keystroke and mouse-button click. Figure 12.21 shows the `InputEvent` methods used to distinguish between mouse-button clicks. Java assumes that every mouse contains a left mouse button. Thus, it is simple to test for a left-mouse-button click. However, users with a one- or two-button mouse must use a combination of pressing keys on the keyboard and clicking the mouse at the same time to simulate the missing buttons on the mouse. In the case of a one- or two-button mouse, this program assumes that the center mouse button is clicked if the user holds the `Alt` key and clicks the left mouse button on a two-button mouse or the only mouse button on a one-button mouse. In the case of a one-button mouse, this program assumes that the right mouse button is clicked if the user holds the `Meta` key and clicks the mouse button.

Method `mouseClicked` (lines 51–73) first captures the coordinates where the event occurred and stores them in instance variables `xPos` and `yPos` of class `MouseDetails`. Lines 56–57 create a string containing the number of mouse clicks (as returned by `MouseEvent` method `getClickCount` at line 57). The nested `if` structure at lines 60–69 uses methods `isMetaDown` and `isAltDown` to determine which mouse button the user clicked and appends an appropriate string to `title` in each case. The resulting string is displayed in the title bar of the window with method `setTitle` (inherited into class `JFrame` from class `Frame`) at line 71. Line 72 calls `repaint` to initiate a call to `paint` to draw a string at the location where the user clicked the mouse.
12.13 Keyboard Event Handling

This section presents the `KeyListener` event-listener interface for handling key events. Key events are generated when keys on the keyboard are pressed and released. A class that implements `KeyListener` must provide definitions for methods `keyPressed`, `keyReleased` and `keyTyped`, each of which receives a `KeyEvent` as its argument. Class `KeyEvent` is a subclass of `InputEvent`. Method `keyPressed` is called in response to pressing any key. Method `keyTyped` is called in response to pressing any key that is not an action key (e.g., an arrow key, `Home`, `End`, `Page Up`, `Page Down`, a function key, `Num Lock`, `Print Screen`, `Scroll Lock`, `Caps Lock` and `Pause`). Method `keyReleased` is called when the key is released after any `keyPressed` or `keyTyped` event.

Figure 12.22 demonstrates the `KeyListener` methods. Class `KeyDemo` implements the `KeyListener` interface, so all three methods are defined in the application.
// set up JTextArea
textArea = new JTextArea( 10, 15 );
textArea.setText( "Press any key on the keyboard..." );
textArea.setEnabled( false );
getContentPane().add( textArea );

// allow frame to process Key events
addKeyListener( this );

setSize( 350, 100 );
setVisible( true );

// handle press of any key
public void keyPressed( KeyEvent event )
{
    line1 = "Key pressed: " +
        event.getKeyText( event.getKeyCode() );
    setLines2and3( event );
}

// handle release of any key
public void keyReleased( KeyEvent event )
{
    line1 = "Key released: " +
        event.getKeyText( event.getKeyCode() );
    setLines2and3( event );
}

// handle press of an action key
public void keyTyped( KeyEvent event )
{
    line1 = "Key typed: " + event.getKeyChar();
    setLines2and3( event );
}

// set second and third lines of output
private void setLines2and3( KeyEvent event )
{
    line2 = "This key is " +
        ( event.isActionKey() ? "an action key" : "not a key" ) +
        "\n";
    String temp =
        event.getKeyModifiersText( event.getModifiers() );
    line3 = "Modifier keys pressed: " +
        ( temp.equals( "" ) ? "none" : temp ) +
        "\n";
    textArea.setText( line1 + "\n" + line2 + "\n" + line3 + "\n" );
}

Fig. 12.22 Demonstrating key event-handling (part 2 of 3).
The constructor (lines 17–32) registers the application to handle its own key events with method `addKeyListener` at line 28. Method `addKeyListener` is defined in class `Component`, so every subclass of `Component` can notify `KeyListener`s of key events for that `Component`.

Line 25 in the constructor adds `JTextAreatextArea` (where the program’s output is displayed) to the content pane. Notice, in the screen captures, that `textArea` occupies the entire window. This is due to the content pane’s default `BorderLayout` (discussed in Section 12.14.2 and demonstrated in Fig. 12.25). When a single `Component` is added to a `BorderLayout`, the `Component` occupies the entire `Container`.

Methods `keyPressed` (lines 35–40) and `keyReleased` (lines 43–48) use `KeyEvent` method `getKeyCode` to get the virtual key code of the key that was pressed. Class `KeyEvent` maintains a set of constants—the virtual key code constants—that represent every key on the keyboard. These constants can be compared with the return value of `getKeyCode` to test for individual keys on the keyboard. The value returned by `getKeyCode` is passed to `KeyEvent` method `getKeyText`, which returns a `String` con-
taining the name of the key that was pressed. For a complete list of virtual key constants, see the on-line documentation for class **KeyEvent** (package `java.awt.event`). Method **keyTyped** (lines 51–55) uses **KeyEvent** method **getKeyChar** to get the Unicode value of the character typed.

All three event handling methods finish by calling method **setLines2and3** (lines 58–72) and passing it the **KeyEvent** object. This method uses **KeyEvent** method **isActionKey** to determine if the key in the event was an action key. Also, **InputEvent** method **getModifiers** is called to determine if any modifier keys (such as **Shift**, **Alt** and **Ctrl**) were pressed when the key event occurred. The result of this method is passed to **KeyEvent** method **getKeyModifiersText**, which produces a string containing the names of the pressed modifier keys.

[Note: If you need to test for a specific key on the keyboard, class **KeyEvent** provides a key constant for every key on the keyboard. These constants can be used from the key event handlers to determine if a particular key was pressed. Also, to determine whether the **Alt**, **Ctrl**, **Meta** and **Shift** keys are pressed individually, **InputEvent** methods **isAltDown**, **isControlDown**, **isMetaDown** and **isShiftDown** each return a boolean indicating if the particular key was pressed during the key event.]

### 12.14 Layout Managers

**Layout managers** are provided to arrange GUI components on a container for presentation purposes. The layout managers provide basic layout capabilities that are easier to use than determining the exact position and size of every GUI component. This enables the programmer to concentrate on the basic “look and feel” and lets the layout managers process most of the layout details.

**Look-and-Feel Observation 12.9**

Most Java programming environments provide GUI design tools that help a programmer graphically design a GUI, then automatically write Java code to create the GUI.

Some GUI designers also allow the programmer to use the layout managers described here and in Chapter 13. Figure 12.23 summarizes the layout managers presented in this chapter. Other layout managers are discussed in Chapter 13.

<table>
<thead>
<tr>
<th>Layout manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FlowLayout</strong></td>
<td>Default for <code>java.awt.Applet</code>, <code>java.awt.Panel</code> and <code>javax.swing.JPanel</code>. Places components sequentially (left to right) in the order they were added. It is also possible to specify the order of the components using the <code>Container</code> method <code>add</code> that takes a <code>Component</code> and an integer index position as arguments.</td>
</tr>
<tr>
<td><strong>BorderLayout</strong></td>
<td>Default for the content panes of <code>JFrames</code> (and other windows) and <code>JApplets</code>. Arranges the components into five areas: North, South, East, West and Center.</td>
</tr>
<tr>
<td><strong>GridLayout</strong></td>
<td>Arranges the components into rows and columns.</td>
</tr>
</tbody>
</table>

*Fig. 12.23* Layout managers.
Most previous applet and application examples in which we created our own GUI used layout manager \texttt{FlowLayout}. Class \texttt{FlowLayout} inherits from class \texttt{Object} and implements interface \texttt{LayoutManager}, which defines the methods a layout manager uses to arrange and size GUI components on a container.

### 12.14.1 FlowLayout

FlowLayout is the most basic layout manager. GUI components are placed on a container from left to right in the order in which they are added to the container. When the edge of the container is reached, components are continued on the next line. Class \texttt{FlowLayout} allows GUI components to be \textit{left-aligned}, \textit{centered} (the default) and \textit{right-aligned}.

The application of Fig. 12.24 creates three \texttt{JButton} objects and adds them to the application, using a \texttt{FlowLayout} layout manager. The components are automatically center-aligned. When the user clicks \texttt{Left}, the alignment for the layout manager is changed to a left-aligned \texttt{FlowLayout}. When the user clicks \texttt{Right}, the alignment for the layout manager is changed to a right-aligned \texttt{FlowLayout}. When the user clicks \texttt{Center}, the alignment for the layout manager is changed to a center-aligned \texttt{FlowLayout}. Each button has its own event handler that is defined with an inner class that implements \texttt{ActionListener}. The sample output windows show each of the \texttt{FlowLayout} alignments. Also, the last sample output window shows the centered alignment after the window has been resized to a smaller width. Notice that the button \texttt{Right} now appears on a new line.

```java
public class FlowLayoutDemo extends JFrame {
    private JButton leftButton, centerButton, rightButton;
    private Container container;
    private FlowLayout layout;

    // set up GUI and register button listeners
    public FlowLayoutDemo() {
        super( "FlowLayout Demo" );
        layout = new FlowLayout();
        // get content pane and set its layout
        container = getContentPane();
        container.setLayout( layout );
    }
```

Fig. 12.24 Program that demonstrates components in \texttt{FlowLayout} (part 1 of 3).
// set up leftButton and register listener
leftButton = new JButton( "Left" );

leftButton.addActionListener(

    // anonymous inner class
    new ActionListener() {

        // process leftButton event
        public void actionPerformed( ActionEvent event )
        {
            layout.setAlignment( FlowLayout.LEFT );
            // re-align attached components
            layout.layoutContainer( container );
        }
    } // end anonymous inner class
); // end call to addActionListener

container.add( leftButton );

// set up centerButton and register listener
centerButton = new JButton( "Center" );

centerButton.addActionListener(

    // anonymous inner class
    new ActionListener() {

        // process centerButton event
        public void actionPerformed( ActionEvent event )
        {
            layout.setAlignment( FlowLayout.CENTER );
            // re-align attached components
            layout.layoutContainer( container );
        }
    } // end anonymous inner class
);

container.add( centerButton );

// set up rightButton and register listener
rightButton = new JButton( "Right" );

rightButton.addActionListener(

    // anonymous inner class
    new ActionListener() {

        // process rightButton event
        public void actionPerformed( ActionEvent event )
        {
            layout.setAlignment( FlowLayout.RIGHT );
            // re-align attached components
            layout.layoutContainer( container );
        }
    } // end anonymous inner class
);

container.add( rightButton );

Fig. 12.24 Program that demonstrates components in FlowLayout (part 2 of 3).
As seen previously, a container’s layout is set with method `setLayout` of class `Container`. Line 25 sets the content pane’s layout manager to the `FlowLayout` defined at line 21. Normally, the layout is set before any GUI components are added to a container.

**Look-and-Feel Observation 12.10**

Each container can have only one layout manager at a time. (Separate containers in the same program can have different layout managers.)
Each button’s `actionPerformed` event handler executes two statements. For example, line 38 in method `actionPerformed` for button `left` uses `FlowLayout` method `setAlignment` to change the alignment for the `FlowLayout` to a left-aligned (`FlowLayout.LEFT`) `FlowLayout`. Line 41 uses `LayoutManager` interface method `layoutContainer` to specify that the content pane should be rearranged based on the adjusted layout.

According to which button was clicked, the `actionPerformed` method for each button sets the `FlowLayout`’s alignment to `FlowLayout.LEFT`, `FlowLayout.CENTER` or `FlowLayout.RIGHT`.

### 12.14.2 BorderLayout

The `BorderLayout` layout manager (the default layout manager for the content pane) arranges components into five regions: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER` (North corresponds to the top of the container). Class `BorderLayout` inherits from `Object` and implements interface `LayoutManager2` (a subinterface of `LayoutManager` that adds several methods for enhanced layout processing).

Up to five components can be added directly to a `BorderLayout`—one for each region. The component placed in each region can be a container to which other components are attached. The components placed in the `NORTH` and `SOUTH` regions extend horizontally to the sides of the container and are as tall as the components placed in those regions. The `EAST` and `WEST` regions expand vertically between the `NORTH` and `SOUTH` regions and are as wide as the components placed in those regions. The component placed in the `CENTER` region expands to take all remaining space in the layout (this is the reason the `JTextArea` in Fig. 12.22 occupies the entire window). If all five regions are occupied, the entire container’s space is covered by GUI components. If the `NORTH` or `SOUTH` region is not occupied, the GUI components in the `EAST`, `CENTER` and `WEST` regions expand vertically to fill the remaining space. If the `EAST` or `WEST` region is not occupied, the GUI component in the `CENTER` region expands horizontally to fill the remaining space. If the `CENTER` region is not occupied, the area is left empty—the other GUI components do not expand to fill the remaining space.

The application of Fig. 12.25 demonstrates the `BorderLayout` layout manager by using five `JButtons`.

```java
// Fig. 12.25: BorderLayoutDemo.java
// Demonstrating BorderLayout.

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

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

public class BorderLayoutDemo extends JFrame
    implements ActionListener {
```
private JButton buttons[];
private String names[] = { "Hide North", "Hide South",
    "Hide East", "Hide West", "Hide Center" };
private BorderLayout layout;

// set up GUI and event handling
public BorderLayoutDemo()
{
    super( "BorderLayout Demo" );
    layout = new BorderLayout( 5, 5 );
    // get content pane and set its layout
    Container container = getContentPane();
    container.setLayout( layout );
    // instantiate button objects
    buttons = new JButton[ names.length ];
    for ( int count = 0; count < names.length; count++ ) {
        buttons[ count ] = new JButton( names[ count ] );
        buttons[ count ].addActionListener( this );
    }
    // place buttons in BorderLayout; order not important
    container.add( buttons[ 0 ], BorderLayout.NORTH );
    container.add( buttons[ 1 ], BorderLayout.SOUTH );
    container.add( buttons[ 2 ], BorderLayout.EAST );
    container.add( buttons[ 3 ], BorderLayout.WEST );
    container.add( buttons[ 4 ], BorderLayout.CENTER );
    setSize( 300, 200 );
    setVisible( true );
}

// handle button events
public void actionPerformed( ActionEvent event )
{
    for ( int count = 0; count < buttons.length; count++ )
    {
        if ( event.getSource() == buttons[ count ] )
            buttons[ count ].setVisible( false );
        else
            buttons[ count ].setVisible( true );
    }
    // re-layout the content pane
    layout.layoutContainer( getContentPane() );
}

// execute application
public static void main( String args[] )
{
    BorderLayoutDemo application = new BorderLayoutDemo();
Fig. 12.25 Demonstrating components in BorderLayout (part 3 of 3).

Line 24 in the constructor defines a `BorderLayout`. The arguments specify the number of pixels between components that are arranged horizontally (`horizontal gap space`) and the number of pixels between components that are arranged vertically (`vertical gap space`), respectively. The default `BorderLayout` constructor supplies 0 pixels of gap space horizontally and vertically. Line 28 uses method `setLayout` to set the content pane’s layout to `layout`.

Adding `Components` to a `BorderLayout` requires a different add method from class `Container`, which takes two arguments—the `Component` to add and the region in which the `Component` will be placed. For example, line 39 specifies that the `buttons[0]` should appear in the `NORTH` position. The components can be added in any order, but only one component can be added to each region.
Look-and-Feel Observation 12.11

If no region is specified when adding a Component to a BorderLayout, it is assumed that the Component should be added to region BorderLayout.CENTER.

Common Programming Error 12.6

Adding more than one component to a particular region in a BorderLayout results in only the last component added being displayed. There is no error message to indicate this problem.

When the user clicks on a particular JButton in the layout, method actionPerformed (lines 50–61) executes. The for loop at lines 52–57 uses an if/else structure to hide the particular JButton that generated the event. Method setVisible (inherited into JButton from class Component) is called with a false argument to hide the JButton. If the current JButton in the array is not the one that generated the event, method setVisible is called with a true argument to ensure that the JButton is displayed on the screen. Line 60 uses LayoutManager method layoutContainer to recalculate the layout of the content pane. Notice in the screen captures of Fig. 12.25 that certain regions in the BorderLayout change shape as JButtons are hidden and displayed in other regions. Try resizing the application window to see how the various regions resize based on the width and height of the window.

12.14.3 GridLayout

The GridLayout layout manager divides the container into a grid so that components can be placed in rows and columns. Class GridLayout inherits directly from class Object and implements interface LayoutManager. Every Component in a GridLayout has the same width and height. Components are added to a GridLayout starting at the top-left cell of the grid and proceeding left-to-right until the row is full. Then the process continues left-to-right on the next row of the grid, etc. Figure 12.26 demonstrates the GridLayout layout manager using six JButtons.

```
// Fig. 12.26: GridLayoutDemo.java
// Demonstrating GridLayout.

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

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

public class GridLayoutDemo extends JFrame
  implements ActionListener {

  private JButton buttons[];
  private String names[] =
    { "one", "two", "three", "four", "five", "six" };
  private boolean toggle = true;

  // Fig. 12.26 Program that demonstrates components in GridLayout.
```
private Container container;
private GridLayout grid1, grid2;

// set up GUI
public GridLayoutDemo()
{
    super( "GridLayout Demo" );

    // set up layouts
    grid1 = new GridLayout( 2, 3, 5, 5 );
    grid2 = new GridLayout( 3, 2 );

    // get content pane and set its layout
    container = getContentPane();
    container.setLayout( grid1 );

    // create and add buttons
    buttons = new JButton[ names.length ];

    for ( int count = 0; count < names.length; count++ ) {
        buttons[ count ] = new JButton( names[ count ] );
        buttons[ count ].addActionListener( this );
        container.add( buttons[ count ] );
    }

    setSize( 300, 150 );
    setVisible( true );
}

// handle button events by toggling between layouts
public void actionPerformed( ActionEvent event )
{
    if ( toggle )
        container.setLayout( grid2 );
    else
        container.setLayout( grid1 );

    toggle = !toggle;   // set toggle to opposite value
    container.validate();
}

// execute application
public static void main( String args[] )
{
    GridLayoutDemo application = new GridLayoutDemo();

    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

}  // end class GridLayoutDemo

Fig. 12.26 Program that demonstrates components in GridLayout.
Lines 27–28 in the constructor define two GridLayout objects. The GridLayout constructor used at line 27 specifies a GridLayout with 2 rows, 3 columns, 5 pixels of horizontal-gap space between Components in the grid and 5 pixels of vertical-gap space between Components in the grid. The GridLayout constructor used at line 28 specifies a GridLayout with 3 rows, 2 columns and no gap space.

The JButton objects in this example initially are arranged using grid1 (set for the content pane at line 32 with method setLayout). The first component is added to the first column of the first row. The next component is added to the second column of the first row, and so on. When a JButton is pressed, method actionPerformed (lines 48–57) is called. Every call to actionPerformed toggles the layout between grid2 and grid1.

Line 56 illustrates another way to relayout a container for which the layout has changed. Container method validate recomputes the container’s layout based on the current layout manager for the Container and the current set of displayed GUI components.

### 12.15 Panels

Complex GUIs (like Fig. 12.1) require that each component be placed in an exact location. They often consist of multiple panels with each panel’s components arranged in a specific layout. Panels are created with class JPanel—a subclass of JComponent. Class JComponent inherits from class java.awt.Container, so every JPanel is a Container. Thus JPanel objects may have components, including other panels, added to them.

The program of Fig. 12.27 demonstrates how a JPanel can be used to create a more complex layout for Components.

```java
// Fig. 12.27: PanelDemo.java
// Using a JPanel to help lay out components.

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

public class PanelDemo extends JFrame {
  private JPanel buttonPanel;

  // Fig. 12.26 Program that demonstrates components in GridLayout.
  
  Fig. 12.26 Program that demonstrates components in GridLayout.

  Fig. 12.27 A JPanel with five JButton objects in a GridLayout attached to the SOUTH region of a BorderLayout (part 1 of 2).  

  © Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
```
```java
private JButton buttons[];

// set up GUI
public PanelDemo()
{
    super( "Panel Demo" );
    // get content pane
    Container container = getContentPane();
    // create buttons array
    buttons = new JButton[ 5 ];
    // set up panel and set its layout
    buttonPanel = new JPanel();
    buttonPanel.setLayout(
        new GridLayout( 1, buttons.length ) );
    // create and add buttons
    for ( int count = 0; count < buttons.length; count++ ) {
        buttons[ count ] =
            new JButton( "Button " + ( count + 1 ) );
        buttonPanel.add( buttons[ count ] );
    }
    container.add( buttonPanel, BorderLayout.SOUTH );
    setSize( 425, 150 );
    setVisible( true );
}

// execute application
public static void main( String args[] )
{
    PanelDemo application = new PanelDemo();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
}  // end class PanelDemo
```

Fig. 12.27 A JPanel with five JButtons in a GridLayout attached to the SOUTH region of a BorderLayout (part 2 of 2).
After `JPanel buttonPanel` is created at line 27, lines 28–29 set `buttonPanel`’s layout to a `GridLayout` of one row and five columns (there are five `JButtons` in array `buttons`). The five `JButtons` in array `buttons` are added to the `JPanel` in the loop with line 35. Notice that the buttons are added directly to the `JPanel`—class `JPanel` does not have a content pane like an applet or a `JFrame`. Line 38 uses the content pane’s default `BorderLayout` to add `buttonPanel` to the `SOUTH` region. Note that the `SOUTH` region is as tall as the buttons on `buttonPanel`. A `JPanel` is sized to the components it contains. As more components are added, the `JPanel` grows (according to the restrictions of its layout manager) to accommodate the components. Resize the window to see how the layout manager affects the size of the `JButtons`.

### 12.16 (Optional Case Study) Thinking About Objects: Use Cases

The previous eight “Thinking About Objects” sections have concentrated on the elevator-simulation model. We have identified and honed the structure and behavior of our system. In this section, we model the interaction between the user and our elevator simulation through the UML use-case diagram, which describes the sets of scenarios that occur between the user and the system.

#### Use-Case Diagrams

When developers begin a project, they rarely start with as detailed a problem statement as the one we provided in Section 2.9. This document and others are the result of the object-oriented analysis (OOA) phase. In this phase you meet with the people who want you to build a system and with the people who will eventually use that system. You use the information gained in these meetings to compile a list of **system requirements**. These requirements guide you and your fellow developers as you design the system. In our case study, the problem statement described the requirements of our elevator simulation in sufficient detail that you did not need to go through an analysis phase. The analysis phase is enormously important—you should consult the references we provide in Section 2.9 to learn more about object-oriented analysis.

The UML provides the **use-case diagram** to facilitate requirements gathering. This diagram models the interactions between the system’s external clients and the **use cases** of the system. Each use case represents a different capability that the system provides to clients. For example, automated teller machines typically have several use cases, including “Deposit Money,” “Withdraw Money” and “Transfer Funds.”

In larger systems, use-case diagrams are indispensable tools that help system designers remain focused on satisfying the users’ needs. The goal of the use-case diagram is to show the kinds of interactions users have with a system without providing the details of those interactions: those details are, of course, provided in other UML diagrams.

Figure 12.28 shows the use-case diagram for our elevator simulation. The stick figure represents an **actor**, which, in turn, represents a set of roles that an **external entity**—such as a person or another system—can play. Consider again our automated teller machine example. The actor is a **BankCustomer** who can deposit, withdraw and transfer funds from the ATM. In this sense, **BankCustomer** is more like a class rather than an object—it is not an actual person, but rather describes the roles that a real person—when playing the part of a **BankCustomer**—can perform while interacting with the ATM (deposit, with-
draw and transfer funds). A person is an external entity that can play the part of a **Bank-Customer**. In the same manner as an object is an instance of a class, a person playing the part of a **BankCustomer** performing one of its roles (such as making a deposit) is an **instance of actor BankCustomer**. For example, when a person named Mary plays the part of a **BankCustomer** making a deposit, Mary—in the role of the depositor—becomes an instance of actor **BankCustomer**. Later in that day, another person named Jon can be another instance of actor **BankCustomer**. In the course of a day, several hundred people might use the ATM machine—some are “depositors”, some are “withdrawers” and some are “transferrers,” but all of these people are instances of actor **BankCustomer**.

The problem statement in our elevator simulation supplies the actors—“The user requires the ability to create a person in the simulation and situate that person on a given floor.” Therefore, the actor of our system is the user who controls the simulation (i.e., the user who clicks the buttons to create new **Person**s in the simulation). An external entity—a real person—plays the part of the user to control the simulation. In our system, the use case is “Create Person,” which encompasses creating a **Person** object, then placing that **Person** on either the first or second **Floor**. Figure 12.28 models one actor called “User.” The actor’s **name** appears underneath the actor.

The **system box** (i.e., the enclosing rectangle in the figure) contains the use cases for the system. Notice that the box is labeled “Elevator Simulation.” This title shows that this use-case model focuses on the one use case that our simulation provides to users (i.e., “Create Person”). The UML models each use case as an oval. The system box for a system with multiple use cases would have one oval per use case.

There is a reasonable alternate view of the use case of our elevator simulation. The problem statement from Section 2.9 mentioned that the company requested the elevator simulation to “determine whether the elevator will meet the company’s needs.” We are designing a simulation of a real-world scenario—the **Person** object in the simulation represents an actual human being using an actual elevator. Thus, we may view the user of the elevator simulation as the user of the elevator. Therefore, specifying a use case from the **Person** object’s perspective helps model how a real person uses a real elevator system. We offer the use case of Fig. 12.29, titled “Relocate Person.” This use case describes the **Person** moving (relocating) to the other **Floor**. (The **Person** travels to the second **Floor** if starting on the first **Floor** and to the first **Floor** if starting on the second **Floor**.) This use case encompasses all actions that the **Person** performs along his or her journey, such as walking across a **Floor** to the **Elevator**, pressing **Buttons** and riding the **Elevator** to the other **Floor**.
We must ensure that our use cases do not model interactions that are too specific between the external client and the system. For example, we do not subdivide each use case into two separate use cases—such as “Create Person on first Floor” and “Create Person on second Floor,” or “Relocate Person to first Floor” and “Relocate Person to second Floor”—because the functionality of such use cases is repetitive (i.e., these seemingly alternative use cases are really the same). Improper and repetitive subdivision of use cases can create problems during implementation. For example, if the designer of an automated teller machine separated its “Withdraw Money” use case into “Withdraw Specific Amounts” use cases (e.g., “Withdraw $1.00,” “Withdraw $2.00,” etc.), there could exist an enormous number of use cases for the system. This would make the implementation tedious. (Our elevator system contains only two floors—separating the use case into two would not cause that much extra work; if our system contained 100 floors, however, creating 100 use cases would be unwieldy.)

**Constructing the Graphical User Interface**

Our simulation implements both the “Create Person” and “Relocate Person” use cases. We have studied the “Relocate Person” use case through the activity diagram of the Person in Fig. 5.29—we implement this use case and the activity diagram in Appendix H when we create class Person. We implement the “Create Person” use case through a graphical user interface (GUI). We implement our GUI in class ElevatorController (Fig. 12.30, line 17), which is a JPanel subclass containing two JButton objects—firstControllerButton (line 21) and secondControllerButton (line 22). Each JButton corresponds to a Floor on which to place a Person. Lines 33–38 instantiate these JButtons and add them to the ElevatorController.

We discuss in Section 13.17 how class ElevatorModel ties together all objects composing our elevator simulation model. Line 25 of class ElevatorController declares a reference to the ElevatorModel, because the ElevatorController allows the user to interact with the model. Lines 42–56 and 60–74 declare two anonymous ActionListener objects and register them with firstFloorControllerButton and secondFloorControllerButton, respectively, for ActionEvents. When the user presses either JButton, lines 49–50 and 67–68 of methods actionPer-

---

1. This approach is feasible with only two Floors. If the building had 100 Floors, we might have opted for the user to specify the desired Floor in a JTextField and press a JButton to process the request.
formed call the ElevatorModel’s method placePersonOnFloor, which instantiates a Person object in the ElevatorModel on the specified Floor. Method placePersonOnFloor takes as an argument a String defined in interface ElevatorConstants (Fig. 12.31). This interface—used by such classes as ElevatorController, ElevatorModel, Elevator, Floor and ElevatorView—provides constants that specify the names of Locations in our simulation.

```java
// ElevatorController.java
// Controller for Elevator Simulation
package com.deitel.jhtp4.elevator.controller;

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

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

// Deitel packages
import com.deitel.jhtp4.elevator.model.*;
import com.deitel.jhtp4.elevator.event.*;
import com.deitel.jhtp4.elevator.ElevatorConstants;

public class ElevatorController extends JPanel
        implements ElevatorConstants {

    // controller contains two JButtons
    private JButton firstControllerButton;
    private JButton secondControllerButton;

    // reference to model
    private ElevatorModel elevatorModel;

    public ElevatorController( ElevatorModel model ) {
        elevatorModel = model;
        setBackground( Color.white );

        // add first button to controller
        firstControllerButton = new JButton( "First Floor" );
        add( firstControllerButton );

        // add second button to controller
        secondControllerButton = new JButton( "Second Floor" );
        add( secondControllerButton );

        // anonymous inner class registers to receive ActionEvents
        // from first Controller JButton
        firstControllerButton.addActionListener( new ActionListener() {
```

Fig. 12.30 Class ElevatorController processes user input (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01
// invoked when a JButton has been pressed
public void actionPerformed(ActionEvent event)
{
    // place Person on first Floor
elevatorModel.placePersonOnFloor(
        FIRST_FLOOR_NAME);

    // disable user input
    firstControllerButton.setEnabled(false);
}
} // end anonymous inner class

// anonymous inner class registers to receive ActionEvents
// from second Controller JButton
secondControllerButton.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent event)
        {
            // place Person on second Floor
elevatorModel.placePersonOnFloor(
                SECOND_FLOOR_NAME);

            // disable user input
            secondControllerButton.setEnabled(false);
        }
    } // end anonymous inner class
); // end anonymous inner class

// anonymous inner class enables user input on Floor if
// Person enters Elevator on that Floor
elevatorModel.addPersonMoveListener(
    new PersonMoveListener() {
        public void personEntered(PersonMoveEvent event)
        {
            // get Floor of departure
            String location =
                event.getLocation().getLocationName();

            // enable first JButton if first Floor departure
            if (location.equals(FIRST_FLOOR_NAME))
                firstControllerButton.setEnabled(true);

            // enable second JButton if second Floor
            else
                secondControllerButton.setEnabled(true);
        }
    } // end method personEntered

Fig. 12.30 Class ElevatorController processes user input (part 2 of 3).
### Fig. 12.30 Class ElevatorController processes user input (part 3 of 3).

Lines 53 and 71 of methods actionPerformed disable the respective JButton to prevent the user from creating more than one Person per Floor. Lines 78–116 of class ElevatorController declare an anonymous PersonMoveListener that registers with the ElevatorModel to reenable the JButton. Method personEntered (lines 82–97) of the PersonMoveListener reenables the JButton associated with the Floor that the Elevator services—after the Person has entered the Elevator, the user may place another Person on the Floor.

We mentioned in Section 9.23 that classes Elevator and Floor inherited attribute capacity from superclass Location—in Appendix H, we were going to use this attribute to prevent more than one Person from occupying a Location. However, the PersonMoveListener’s method personEntered in class ElevatorController prevents the user from creating more than one Person per Floor. Therefore, we have negated the need for attribute capacity in class Location. Figure 12.32 is the modified class diagram of Fig. 9.18 removing this attribute.

### Fig. 12.31 Interface ElevatorConstants provides Location name constants.

```java
// ElevatorConstants.java
// Constants used between ElevatorModel and ElevatorView
package com.deitel.jhtp4.elevator;

public interface ElevatorConstants {

    public static final String FIRST_FLOOR_NAME = "firstFloor";
    public static final String SECOND_FLOOR_NAME = "secondFloor";
    public static final String ELEVATOR_NAME = "elevator";
}
```
In this section we mentioned that the goal of object-oriented analysis is to produce a system-requirements document. We introduced the UML use-case diagram that facilitates gathering system requirements, and we examined the two use cases in our elevator simulation. We implemented our simulator’s Graphical User Interface in Java.

This section concludes the discussion on the interaction between the user and the simulation model. In “Thinking About Objects” Section 13.17, we integrate class ElevatorController with the rest of the simulation. We also introduce the UML Component diagram, which models the .class, .java, image and sound files that comprise our system.

**SUMMARY**

- A graphical user interface (GUI) presents a pictorial interface to a program. A GUI (pronounced “GOO-EE”) gives a program a distinctive “look” and “feel.”

- By providing different applications with a consistent set of intuitive user interface components, GUIs allow the user to spend more time using the program in a productive manner.

- GUIs are built from GUI components (sometimes called controls or widgets). A GUI component is a visual object with which the user interacts via the mouse or the keyboard.

- Swing GUI components are defined in package javax.swing. Swing components are written, manipulated and displayed completely in Java.
• The original GUI components from the Abstract Windowing Toolkit package `java.awt` are tied directly to the local platform’s graphical user interface capabilities.

• Swing components are lightweight components. AWT components are tied to the local platform and are called heavyweight components—they must rely on the local platform’s windowing system to determine their functionality and their look and feel.

• Several Swing GUI components are heavyweight GUI components: in particular, subclasses of `java.awt.Window` (such as `JFrame`) that display windows on the screen. Heavyweight Swing GUI components are less flexible than lightweight components.

• Much of each Swing GUI component’s functionality is inherited from classes `Component`, `Container` and `JComponent` (the superclass to most Swing components).

• A Container is an area where components can be placed.

• `JLabel` s provide text instructions or information on a GUI.

• `JComponent` method `setToolTipText` specifies the tool tip that is displayed automatically when the user positions the mouse cursor over a `JComponent` in the GUI.

• Many Swing components can display images by specifying an `Icon` as an argument to their constructor or by using a method `setIcon`.

• Class `ImageIcon` (package `javax.swing`) supports several image formats, including Portable Network Graphics (PNG), Graphics Interchange Format (GIF) and Joint Photographic Experts Group (JPEG).

• Interface `SwingConstants` (package `javax.swing`) defines a set of common integer constants (such as `SwingConstants.LEFT`) that are used with many Swing components.

• By default, the text of a `JComponent` appears to the right of the image when the `JComponent` contains both text and an image.

• The horizontal and vertical alignments of a `JLabel` can be set with methods `setHorizontalAlignment` and `setVerticalAlignment`. Method `setText` sets the text displayed on the label. Method `getText` retrieves the current text displayed on a label. Methods `setHorizontalTextPosition` and `setVerticalTextPosition` specify the text position in a label.

• `JComponent` method `setIcon` sets the `Icon` displayed on a `JComponent`. Method `getIcon` retrieves the current `Icon` displayed on a `JComponent`.

• GUIs generate events when the user interacts with the GUI. Information about a GUI event is stored in an object of a class that extends `AWTEvent`.

• To process an event, the programmer must register an event listener and implement one or more event handlers.

• The use of event listeners in event handling is known as the delegation event model—the processing of an event is delegated to a particular object in the program.

• When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener’s appropriate event handling method.

• `JTextFields` and `JPasswordField` s are single-line areas in which text can be entered by the user from the keyboard or text can simply be displayed. A `JPasswordField` shows that a character was typed as the user enters characters, but automatically hides the characters.

• When the user types data into a `JTextField` or `JPasswordField` and presses the Enter key, an `ActionEvent` occurs.

• `JTextComponent` method `setEditable` determines whether the user can modify the text in a `JTextComponent`.

• `JPasswordField` method `getPassword` returns the password as an array of type `char`.
• Every JComponent contains an object of class EventListenerList (package javax.swing.event) called listenerList in which all registered listeners are stored.

• Every JComponent supports several different event types, including mouse events, key events and others. When an event occurs, the event is dispatched only to the event listeners of the appropriate type. Each event type has a corresponding event-listener interface.

• When an event is generated by a user interaction with a component, the component is handed a unique event ID specifying the event type. The GUI component uses the event ID to decide the type of listener to which the event should be dispatched and the event handler method to call.

• A JButton generates an ActionEvent when the user clicks the button with the mouse.

• An AbstractButton can have a rollover Icon that is displayed when the mouse is positioned over the button. The icon changes as the mouse moves in and out of the button’s area on the screen. AbstractButton method setRolloverIcon specifies the image displayed on a button when the user positions the mouse over the button.

• The Swing GUI components contain three state button types—JToggleButton, JCheckBox and JRadioButton—that have on/off or true/false values. Classes JCheckBox and JRadioButton are subclasses of JToggleButton.

• When the user clicks a JCheckBox, an ItemEvent is generated that can be handled by an ItemListener. ItemListeners must define method itemStateChanged. ItemEvent method getStateChange determines the state of a JToggleButton.

• JRadioButtons are similar to JCheckBoxes in that they have two states—selected and not selected (also called deselected). JRadioButtons normally appear as a group in which only one radio button can be selected at a time.

• The logical relationship between radio buttons is maintained by a ButtonGroup object.

• The JRadioButton constructor supplies the label that appears to the right of the JRadioButton by default and the initial state of the JRadioButton. A true second argument indicates that the JRadioButton should appear selected when it is displayed.

• JRadioButtons generate ItemEvents when they are clicked.

• ButtonGroup method add associates a JRadioButton with a ButtonGroup. If more than one selected JRadioButton object is added to the group, the last selected JRadioButton added will be selected when the GUI is displayed.

• A JComboBox (sometimes called a drop-down list) provides a list of items from which the user can make a selection. JComboBoxes generate ItemEvents. A numeric index keeps track of the ordering of items in a JComboBox. The first item is added at index 0, the next item is added at index 1 and so forth. The first item added to a JComboBox appears as the currently selected item when the JComboBox is displayed. JComboBox method getSelectedIndex returns the index number of the selected item.

• A JList displays a series of items from which the user may select one or more items. Class JList supports single- and multiple-selection lists. When an item is clicked in a JList, a ListSelectionEvent occurs.

• JList method setVisibleRowCount determines the number of items that are visible in the list. Method setSelectionMode specifies the selection mode for the list.

• Class JList does not automatically provide a scrollbar if there are more items in the list than the number of visible rows. A JScrollPane object is used to provide the automatic scrolling capability for a JList.

• A SINGLE_INTERVAL_SELECTION list allows selection of a contiguous range of items by clicking the first item, then holding the Shift key while clicking the last item to select in the range.
• A **MULTIPLE_INTERVAL_SELECTION** list allows continuous range selection as described for a **SINGLE_INTERVAL_SELECTION** list and allows miscellaneous items to be selected by holding the Ctrl key while clicking each item to select.

• **JList** method **setFixedCellHeight** specifies the height in pixels of each item in a **JList**. Method **setFixedCellWidth** sets the width in pixels of a **JList**.

• Normally, an event generated by another GUI component (known as an external event) specifies when the multiple selections in a **JList** should be processed.

• **JList** method **setListData** sets the items displayed in a **JList**. Method **getSelectedValues** returns the selected items as an array of **Objects**.

• Mouse events can be trapped for any GUI component that derives from **java.awt.Component** using **MouseListener**s and **MouseMotionListener**s.

• Each mouse event handling method takes as its argument a **MouseEvent** object containing information about the mouse event and the location where the event occurred.

• Methods **addMouseListener** and **addMouseMotionListener** are **Component** methods used to register mouse event listeners for an object of any class that extends **Component**.

• Many of the event-listener interfaces provide multiple methods. For each, there is a corresponding event-listener adapter class that provides a default implementation of every method in the interface. The programmer can extend the adapter class to inherit the default implementation of every method and simply override the method or methods needed for event handling in the program.

• **MouseEvent** method **getClickCount** returns the number of mouse clicks.

• **InputEvent** methods **isMetaDown** and **isAltDown** are used to determine which mouse button the user clicked.

• **KeyListener**s handle key events that are generated when keys on the keyboard are pressed and released. A **KeyListener** must provide definitions for methods **keyPressed**, **keyReleased** and **keyTyped**, each of which receives a **KeyEvent** as its argument.

• Method **keyPressed** is called in response to pressing any key. Method **keyTyped** is called in response to pressing any key that is not an action key (i.e., an arrow key, **Home**, **End**, **Page Up**, **Page Down**, a function key, **Num Lock**, **Print Screen**, **Scroll Lock**, **Caps Lock** and **Pause**). Method **keyReleased** is called when the key is released after any **keyPressed** or **keyTyped** event.

• **KeyEvent** method **getKeyCode** gets the virtual key code of the key that was pressed. Class **KeyEvent** maintains a set of virtual key code constants that represent every key on the keyboard.

• **KeyEvent** method **getKeyText** returns a **String** containing the name of the key that corresponds to its virtual key code argument. Method **getKeyChar** gets the Unicode value of the character typed. Method **isActionKey** determines if the key in the event was an action key.

• **InputEvent** method **getModifiers** determines if any modifier keys (such as **Shift**, **Alt** and **Ctrl**) were pressed when the key event occurred. **KeyEvent** method **getKeyModifiersText** produces a string containing the names of the pressed modifier keys.

• Layout managers arrange GUI components on a container for presentation purposes.

• **FlowLayout** lays out components from left to right in the order in which they are added to the container. When the edge of the container is reached, components are continued on the next line.

• **FlowLayout** method **setAlignment** changes the alignment for the **FlowLayout** to **FlowLayout.LEFT**, **FlowLayout.CENTER** or **FlowLayout.RIGHT**.

• The **BorderLayout** layout manager arranges components into five regions: North, South, East, West and Center. One component can be added to each region.
• **LayoutManager** method `layoutContainer` recalculates the layout of its `Container` argument.

• The `GridLayout` layout manager divides the container into a grid of rows and columns. Components are added to a `GridLayout` starting at the top-left cell and proceeding from left to right until the row is full. Then the process continues from left to right on the next row of the grid, and so on.

• **Container** method `validate` recomputes the container’s layout based on the current layout manager for the `Container` and the current set of displayed GUI components.

• Panels are created with class `JPanel`, which inherits from class `JComponent`. JPanels may have components, including other panels, added to them.

### TERMINOLOGY

- `.gif` file name extension
- `.jpg` file name extension
- “listen” for an event
- Abstract Windowing Toolkit
- AbstractButton class
- ActionEvent class
- ActionListener interface
- actionPerformed method
- adapter class
- `add` method of ButtonGroup
- `add` method of class Container
- `addItemListener` method
- `addKeyListener` method
- `addListSelectionListener` method
- `addMouseListener` method
- `addMouseMotionListener` method
- assistive technologies
- BorderLayout class
- BorderLayout.CENTER
- BorderLayout.EAST
- BorderLayout.NORTH
- BorderLayout.SOUTH
- BorderLayout.WEST
- button
- button label
- ButtonGroup class
- centered
- check box
- check box label
- command button
- Component class
- ComponentAdapter class
- ComponentListener interface
- Container class
- ContainerAdapter class
- ContainerListener interface
- control
- delegation event model
- dispatch an event
- dragging
- drop-down list
- event
- event driven
- event handler
- event ID
- event listener
- event-listener interface
- EventListenerList class
- EventObject class
- FlowLayout class
- FlowLayout.CENTER
- FlowLayout.LEFT
- FlowLayout.RIGHT
- focus
- FocusAdapter class
- FocusListener interface
- Font.BOLD
- Font.ITALIC
- Font.PLAIN
- getActionCommand method
- getClickCount method
- getIcon method
- getKeyChar method of KeyEvent
- getKeyCode method of KeyCode
- getKeyModifiersText method
- getKeyText method of KeyEvent
- getModifiers method of InputEvent
- getPassword method of JPasswordField
- getSelectedIndex method of JComboBox
- getSelectedIndex method of JList
- getSelectedValues method of JList
- getSource method of MouseEvent
- getStateChange method of ItemEvent
- getText method of JLabel
- getX method of MouseEvent
- getY method of MouseEvent
Graphics Interchange Format (GIF)

**GridLayout** class
GUI component
heavyweight component
horizontal gap space
**Icon** interface
**ImageIcon** class
**InputEvent** class
**isActionKey** method of **KeyEvent**
**isAltDown** method of **InputEvent**
**isMetaDown** method of **InputEvent**
**ItemEvent** class
**ItemEvent**.DESELECTED
**ItemEvent**.SELECTED
**ItemListener** interface
**itemStateChanged** method
java.awt package
java.awt.event package
javax.swing package
**JButton** class
**JCheckBox** class
right-aligned
**JComboBox** class
rollover icon
**JComponent** class
scroll arrow
**JLabel** class
scroll box
**JList** class
Joint Photographic Experts Group (JPEG)
**JPanel** class
**JPasswordField** class
left aligned
**JRadioButton** class
right-justified shortcut key (mnemonics)
**JScrollPane** class
**JTextComponent** class
**JTextField** class
**JToggleButton** class
left aligned
**KeyAdapter** class
**KeyListener** interface
**KeyPress** method of **KeyListener**
**keyReleased** method of **KeyListener**
**keyTyped** method of **KeyListener**
label
layout manager
**layoutContainer** method
**LayoutManager** interface
left aligned
left justified
lightweight component
**ListSelectionEvent** class
**ListSelectionListener** interface
**ListSelectionModel** interface
look and feel
menu
menu bar
**MouseAdapter** class
**mouseClicked** method
**mouseDragged** method
**mouseEntered** method
**MouseEvent** class
**mouseExited** method
**MouseListener** interface
**MouseMotionAdapter** class
**MouseMotionListener** interface
**mouseMoved** method
**mousePressed** method
**mouseReleased** method
multiple-selection list
password
pluggable look and feel
radio button
read-only text
register an event listener
right-aligned
rollover icon
scroll arrow
scroll bar
scrollbar
selection mode
**setAlignment** method
**setBackground** method
**setEditable** method
**setFixedCellHeight** method
**setFixedCellWidth** method
**setHorizontalAlignment** method
**setHorizontalTextPosition** method
**setIcon** method
**setListData** method of **JList**
**setMaxRowCount** method
**setRolloverIcon** method
**setSelectionMode** method
**setToolTipText** method
**setVerticalAlignment** method
**setVerticalTextPosition** method
**setVisible** method
**setVisibleRowCount** method
shortcut key (mnemonics)
single-selection list
**SwingConstants** interface
tool tips
toolbar
SELF-REVIEW EXERCISES

12.1 Fill in the blanks in each of the following statements:
   a) Method _______ is called when the mouse is moved and an event listener is registered to handle the event.
   b) Text that cannot be modified by the user is called _______ text.
   c) A _______ arranges GUI components on a Container.
   d) The add method for attaching GUI components is a _______ class method.
   e) GUI is an acronym for _______.
   f) Method _______ is used to set the layout manager for a container.
   g) A mouseDragged method call is preceded by a _______ method call and followed by a _______ method call.

12.2 State whether each of the following is true or false. If false, explain why.
   a) BorderLayout is the default layout manager for a content pane.
   b) When the mouse cursor is moved into the bounds of a GUI component, method mouse- Over is called.
   c) A JPanel cannot be added to another JPanel.
   d) In a BorderLayout, two buttons added to the NORTH region will be placed side by side.
   e) When one is using BorderLayout, a maximum of five components may be used.

12.3 Find the error(s) in each of the following and explain how to correct it (them).
   a) buttonName = JButton( "Caption" );
   b) JLabel aLabel, JLabel;     // create references
   c) txtField = new JTextField( 50, "Default Text" );
   d) Container container = getContentPane();
      setLayout( new BorderLayout() );
      button1 = new JButton( "North Star" );
      button2 = new JButton( "South Pole" );
      container.add( button1 );
      container.add( button2 );

ANSWERS TO SELF-REVIEW EXERCISES

12.1 a) mouseMoved. b) uneditable (read-only). c) layout manager. d) Container. e) graphical user interface. f) setLayout. g) mousePressed, mouseReleased.

12.2 a) True.
   b) False. Method mouseEntered is called.
   c) False. A JPanel cannot be added to another JPanel because JPanel derives indirectly from Component. Therefore, a JPanel is a Component. Any Component can be added to a Container.
   d) False. Only the last button added will be displayed. Remember that only one component can be added to each region in a BorderLayout.
   e) True.

12.3 a) new is needed to instantiate the object.
   b) JLabel is a class name and cannot be used as a variable name.
c) The arguments passed to the constructor are reversed. The `String` must be passed first.

d) `BorderLayout` has been set and components are being added without specifying the region. Proper `add` statements might be

```java
container.add( button1, BorderLayout.NORTH );
container.add( button2, BorderLayout.SOUTH );
```

**EXERCISES**

12.4 Fill in the blanks in each of the following statements:

a) The `JTextField` class inherits directly from __________.

b) The layout managers discussed in this chapter are __________, __________ and __________.

c) `Container` method __________ attaches a GUI component to a container.

d) Method __________ is called when a mouse button is released (without moving the mouse).

e) The __________ class is used to create a group of `JRadioButton`.

12.5 State whether each of the following is true or false. If false, explain why.

a) Only one layout manager can be used per `Container`.

b) GUI components can be added to a `Container` in any order in a `BorderLayout`.

c) `JRadioButtons` provide a series of mutually exclusive options (only one can be `true` at a time).

d) `Graphics` method `setFont` is used to set the font for text fields.

e) A `JList` displays a scrollbar if there are more items in the list than can be displayed.

f) A `Mouse` object contains a method called `mouseDragged`.

12.6 State whether each of the following is true or false. If false, explain why.

a) A `JApplet` does not have a content pane.

b) A `JPanel` is a `JComponent`.

c) A `JPanel` is a `Component`.

d) A `JLabel` is a `Container`.

e) A `JList` is a `JPanel`.

f) An `AbstractButton` is a ` JButton`.

g) A `JTextField` is an `Object`.

h) `ButtonGroup` inherits from `JComponent`.

12.7 Find any error(s) in each of the following and explain how to correct it (them).

a) `import javax.swing.*`       // include swing package

b) `panelObject.GridLayout( 8, 8 );` // set GridLayout

c) `container.setLayout(
 new FlowLayout( FlowLayout.DEFAULT ) );`

d) `container.add( eastButton, EAST );` // `BorderLayout`

12.8 Create the following GUI. You do not have to provide any functionality.
12.9 Create the following GUI. You do not have to provide any functionality.

![Calculator GUI](image)

12.10 Create the following GUI. You do not have to provide any functionality.

![ColorSelect GUI](image)

12.11 Create the following GUI. You do not have to provide any functionality.

![Printer GUI](image)

12.12 Write a temperature conversion program that converts from Fahrenheit to Celsius. The Fahrenheit temperature should be entered from the keyboard (via a JTextField). A JLabel should be used to display the converted temperature. Use the following formula for the conversion:

\[
Celsius = \frac{5}{9} \times (Fahrenheit - 32)
\]

12.13 Enhance the temperature conversion program of Exercise 12.12 by adding the Kelvin temperature scale. The program should also allow the user to make conversions between any two scales. Use the following formula for the conversion between Kelvin and Celsius (in addition to the formula in Exercise 12.12):

\[
Kelvin = Celsius + 273
\]

12.14 Write an application that allows the user to draw a rectangle by dragging the mouse on the application window. The upper-left coordinate should be the location where the user presses the mouse button, and the lower-right coordinate should be the location where the user releases the mouse button. Also display the area of the rectangle in a JLabel in the SOUTH region of a BorderLayout. Use the following formula for the area:

\[
area = width \times height
\]

12.15 Modify the program of Exercise 12.14 to draw different shapes. The user should be allowed to choose from an oval, an arc, a line, a rectangle with rounded corners and a predefined polygon. Also display the mouse coordinates in the status bar.
12.16 Write a program that will allow the user to draw a shape with the mouse. The shape to draw should be determined by a `KeyEvent` using the following keys: `c` draws a circle, `o` draws an oval, `r` draws a rectangle and `l` draws a line. The size and placement of the shape should be determined by the `mousePressed` and `mouseReleased` events. Display the name of the current shape in a `JLabel` in the `SOUTH` region of a `BorderLayout`. The initial shape should default to a circle.

12.17 Create an application that enables the user to paint a picture. The user should be able to choose the shape to draw, the color in which the shape should appear and whether the shape should be filled with color. Use the graphical user interface components we discussed in this chapter, such as `JComboBoxes`, `JRadioButtons` and `J_checkboxes`, to allow the user to select various options. The program should provide a `JButton` object that allows the user to erase the window.

12.18 Write a program that uses `System.out.println` statements to print out events as they occur. Provide a `JComboBox` with a minimum of four items. The user should be able to choose an event to “monitor” from the `JComboBox`. When that particular event occurs, display information about the event in a message dialog box. Use method `toString` on the event object to convert it to a string representation.

12.19 Write a program that draws a square. As the mouse moves over the drawing area, repaint the square with the upper-left corner of the square following the exact path of the mouse cursor.

12.20 Modify the program of Fig. 12.19 to incorporate colors. In a separate window, provide a “toolbar” of `JRadioButton` objects that lists the following six colors: red, black, magenta, blue, green and yellow. The toolbar should be implemented as a subclass of `JFrame` called `ToolBarWindow` and should consist of six buttons, each with the appropriate color name. When a new color is selected, drawing should occur in the new color. Determine the currently selected color in the `mousePressed` event handler of the main window by calling a public method `getCurrentColor` on the `ToolBarWindow`. [Note: In Chapter 13, we discuss how to combine GUI components and drawing, using separate `JPanels` for each. This provides programs with more flexibility in laying out the components and drawing.]

12.21 Write a program that plays “guess the number” as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1–1000. The program then displays in a label:

```
I have a number between 1 and 1000 can you guess my number?
Please enter your first guess.
```

A `JTextField` should be used to input the guess. As each guess is input the background color should change to either red or blue. Red indicates that the user is getting “warmer” and blue indicates that the user is getting “colder.” A `JLabel` should display either “Too High” or “Too Low” to help the user zero in on the correct answer. When the user gets the correct answer, “Correct!” should be displayed and the `JTextField` used for input should be changed to uneditable. A `JButton` should be provided to allow the user to play the game again. When the `JButton` is clicked, a new random number should be generated and the input `JTextField` changed to editable.

12.22 It is often useful to display the events that occur during the execution of a program to help understand when the events occur and how they are generated. Write a program that enables the user to generate and process every event discussed in this chapter. The program should provide methods from the `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` and `KeyListener` interfaces to display messages when the events occur. Use method `toString` to convert the event objects received in each event handler into a `String` that can be displayed. Method `toString` creates a `String` containing all the information in the event object.
12.23 Modify your solution to Exercise 12.17 to enable the user to select a font and a font size, then type text into a JTextField. When the user presses Enter, the text should be displayed on the background in the chosen font and size. Modify the program further to allow the user to specify the exact position at which the text should be displayed.

12.24 Write a program that allows the user to select a shape from a JComboBox, then draws that shape 20 times with random locations and dimensions in method paint. The first item in the JComboBox should be the default shape that is displayed the first time paint is called.

12.25 Modify Exercise 12.24 to draw each of the 20 randomly sized shapes in a randomly selected color. Use all 13 predefined Color objects in an array of Colors.

12.26 Modify Exercise 12.25 to allow the user to select the color in which shapes should be drawn from a JColorChooser dialog.

12.27 Write a program using methods from interface MouseListener that allows the user to press the mouse button, drag the mouse and release the mouse button. When the mouse is released, draw a rectangle with the appropriate upper-left corner, width and height. (Hint: The mousePressed method should capture the set of coordinates at which the user presses and holds the mouse button initially, and the mouseReleased method should capture the set of coordinates at which the user releases the mouse button. Both methods should store the appropriate coordinate values. All calculations of the width, height and upper-left corner should be performed by the paint method before the shape is drawn.)

12.28 Modify Exercise 12.27 to provided a “rubber-banding” effect. As the user drags the mouse, the user should be able to see the current size of the rectangle to know exactly what the rectangle will look like when the mouse button is released. (Hint: Method mouseDragged should perform the same tasks as mouseReleased.)

12.29 Modify Exercise 12.28 to allow the user to select which shape to draw. A JComboBox should provide options including at least rectangle, oval, line and rounded rectangle.

12.30 Modify Exercise 12.29 to allow the user to select the drawing color from a JColorChooser dialog box.

12.31 Modify Exercise 12.30 to allow the user to specify whether a shape should be filled or empty when it is drawn. The user should click a JCheckBox to indicate filled or empty.

12.32 (Painting program) Using the techniques of Exercise 9.28–Exercise 9.29 and Exercise 12.27–Exercise 12.30 and the graphics techniques of Chapter 11, rewrite Exercise 12.31 to allow the user to draw multiple shapes and store each shape in an array of shapes. (If you feel ambitious, investigate the capabilities of class Vector in Chapter 20.) For this program, create your own classes (like those in the class hierarchy described in Exercise 9.28–Exercise 9.29) from which objects will be created to store each shape the user draws. The classes should store the location, dimensions and color of each shape and should indicate whether the shape is filled or unfilled. Your classes should all derive from a class called MyShape that has all the common features of every shape type. Every subclass of MyShape should have its own method draw, which returns void and receives a Graphics object as its argument. When the application window’s paint method is called, it should walk through the array of shapes and display each shape by polymorphically calling the shape’s draw method (passing the Graphics object as an argument). Each shape’s draw method should know how to draw the shape. As a minimum, your program should provide the following classes: MyLine, MyOval, MyRect, MyRoundRect. Design the class hierarchy for maximum software reuse, and place all your classes in the package shapes. Import this package into your program.

12.33 Modify Exercise 12.32 to provide an Undo button that can be used repeatedly to undo the last painting operation. If there are no shapes in the array of shapes, the Undo button should be disabled.
Objectives

- To create and manipulate text areas, sliders, menus, popup menus and windows.
- To be able to create customized `JPanel` objects.
- To be able to create a program that can execute as either an applet or an application.
- To be able to change the look-and-feel of a GUI, using Swing’s pluggable look-and-feel (PLAF).
- To be able to create a multiple document interface with `JDesktopPane` and `JInternalFrame`.
- To be able to use advanced layout managers.

*I claim not to have controlled events, but confess plainly that events have controlled me.*
Abraham Lincoln

*A good symbol is the best argument, and is a missionary to persuade thousands.*
Ralph Waldo Emerson

*Capture its reality in paint!*
Paul Cézanne
13.1 Introduction

In this chapter, we continue our study of GUIs. We discuss more advanced components and layout managers and lay the groundwork for building complex GUIs.

We begin our discussion with another text-based GUI component—*JTextArea*—which allows multiple lines of text to be displayed or input. We continue with two examples of customizing class *JPanel* in which we discuss issues that relate to painting on Swing GUI components. Next, we illustrate how to design a Java program that can execute as both an applet and an application. An important aspect of any complete GUI is a system of *menus* that enable the user to effectively perform tasks in the program. The next two examples discuss how to create and use menus. The look-and-feel of a Swing GUI can be uniform across all platforms on which the Java program is executed, or the GUI can be...
customized by using Swing’s *pluggable look-and-feel (PLAF)*. The next example illustrates how to change between Swing’s default *metal look-and-feel*, a look-and-feel that simulates *Motif* (a popular UNIX look-and-feel) and one that simulates Microsoft’s Windows look-and-feel. Many of today’s applications use a *multiple document interface (MDI)*, i.e., a main window (often called the *parent window*) containing other windows (often called *child windows*) to manage several open *documents* in parallel. For example, many e-mail programs allow you to have several e-mail windows open at the same time so you can compose and/or read multiple e-mail messages. The next example discusses Swing’s classes that provide support for creating multiple document interfaces. Finally, the chapter finishes with a series of examples discussing several advanced layout managers for organizing graphical user interfaces.

Swing is a large and complex topic. There are many more GUI components and capabilities than can be presented here. Several more Swing GUI components are introduced in the remaining chapters of this book as they are needed. Our book *Advanced Java 2 Platform How to Program* discusses other, more advanced Swing components and capabilities.

### 13.2 JTextArea

*JTextArea* provides an area for manipulating multiple lines of text. Like class *JTextField*, class *JTextArea* inherits from *JTextComponent*, which defines common methods for *JTextFields*, *JTextAreas* and several other text-based GUI components.

The application of Fig. 13.1 demonstrates *JTextArea*. One *JTextArea* displays text that the user can select. The other *JTextArea* is uneditable. Its purpose is to display the text the user selected in the first *JTextArea*. *JTextAreas* do not have action events like *JTextFields*. Often, an *external event*—an event generated by a different GUI component—indicates when to process the text in a *JTextArea*. For example, when typing an e-mail message, you often click a *Send* button to take the text of the message and send it to the recipient. Similarly, when editing a document in a word processor, you normally save the file by selecting a menu item called *Save* or *Save As*... In this program, the button *Copy >>>* generates the external event that copies the selected text in the left *JTextArea* and displays it in the right *JTextArea*.

```java
// Fig. 13.1: TextAreaDemo.java
// Copying selected text from one text area to another.

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

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

public class TextAreaDemo extends JFrame {
    private JTextArea textArea1, textArea2;
    private JButton copyButton;
```

Fig. 13.1 Copying selected text from one text area to another (part 1 of 3).
// set up GUI
public TextAreaDemo()
{
    super( "TextArea Demo" );

    Box box = Box.createHorizontalBox();

    String string = "This is a demo string to
    illustrate copying text
    from one TextArea to 
    another TextArea using an\n    external event\n";

    // set up textArea1
textArea1 = new JTextArea( string, 10, 15 );
    box.add( new JScrollPane( textArea1 ) );

    // set up copyButton
copyButton = new JButton( "Copy >>>" );
copyButton.addActionListener(
        // anonymous inner class to handle copyButton event
        new ActionListener() {
            // set text in textArea2 to selected
            // text from textArea1
            public void actionPerformed( ActionEvent event )
            {
                textArea2.setText( textArea1.getSelectedText() );
            }
        }  // end anonymous inner class
    ); // end call to addActionListener

    box.add( copyButton );

    // set up textArea2
textArea2 = new JTextArea( 10, 15 );
textArea2.setEditable( false );
    box.add( new JScrollPane( textArea2 ) );

    // add box to content pane
    Container container = getContentPane();
    container.add( box );   // place in BorderLayout.CENTER

    setSize( 425, 200 );
    setVisible( true );
}

// execute application
public static void main( String args[] )
{
    TextAreaDemo application = new TextAreaDemo();
Look-and-Feel Observation 13.1

 Often an external event determines when the text in a JTextArea should be processed.

In the constructor method (lines 16–62), line 20 creates a Box container (package javax.swing) for organizing the GUI components. Class Box is a subclass of Container that uses a BoxLayout layout manager to arrange the GUI components either horizontally or vertically. Section 13.13 discusses BoxLayout in detail. Class Box provides static method createHorizontalBox to create a Box that arranges components from left to right in the order that the components are attached.

The application instantiates JTextArea objects textArea1 (line 28) and textArea2 (line 52). Each JTextArea has 10 visible rows and 15 visible columns. Line 28 specifies that string should be displayed as the default JTextArea content. A JTextArea does not provide scrollbars if it cannot display its complete contents. For this reason, line 29 creates a JScrollPane object, initializes it with textArea1 and attaches it to container box. By default, horizontal and vertical scrollbars will appear as necessary.

Lines 32–49 instantiate JButton object copyButton with the label “Copy >>>,” create an anonymous inner class to handle copyButton’s ActionEvent and add copyButton to container box. This button provides the external event that determines when the program should copy the selected text in textArea1 to textArea2. When the user clicks copyButton, line 42 in actionPerformed indicates that method
**getSelectedText** (inherited into JTextArea from JTextComponent) should return the selected text from textarea1. The user selects text by dragging the mouse over the desired text to highlight it. Method setText changes the text in textarea2 to the String that method getSelectedText returns.

Lines 52–54 create textarea2 and add it to container box. Lines 57–58 obtain the content pane for the window and add box to the content pane. Remember that the default layout of the content pane is a BorderLayout and that the add method attaches its argument to the CENTER of the BorderLayout if method add does not specify the region.

It is sometimes desirable when text reaches the right side of a JTextArea to have the text wrap to the next line. This is referred to as automatic word wrap.

**Look-and-Feel Observation 13.2**

To provide automatic word wrap functionality for a JTextArea, invoke JTextArea method setLineWrap with a true argument.

This example uses a JScrollPane to provide scrolling functionality for a JTextArea. By default, JScrollPane provides scrollbars only if they are required. You can set the horizontal and vertical scrollbar policies for the JScrollPane when a JScrollPane is constructed or with methods setHorizontalScrollBarPolicy and setVerticalScrollBarPolicy of class JScrollPane at any time. Class JScrollPane provides the constants

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

to indicate that a scrollbar should always appear, constants

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

to indicate that a scrollbar should appear only if necessary, and constants

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

to indicate that a scrollbar should never appear. If the horizontal scrollbar policy is set to JScrollPane.HORIZONTAL_SCROLLBAR_NEVER, a JTextArea attached to the JScrollPane will exhibit automatic word wrap behavior.

### 13.3 Creating a Customized Subclass of JPanel

In Chapter 12, we saw that Panels can aggregate a set of GUI components for layout purposes. Panels are quite flexible. Some of their many uses include creating dedicated drawing areas and creating areas that receive mouse events. Programs often extend class JPanel to create new components. Our next example uses a JPanel to create a dedicated drawing area. Dedicated drawing areas help separate drawing from the rest of your graphical user interface. This can be beneficial in Swing graphical user interfaces. If graphics and Swing GUI components are not displayed in the correct order, it is possible that the GUI components will not display correctly. For example, to ensure that graphics and GUI both display correctly, we can separate the GUI and the graphics by creating dedicated drawing areas as subclasses of JPanel.
Look-and-Feel Observation 13.3

Combining graphics and Swing GUI components can lead to incorrect display of the graphics, the GUI components or both. Using JPanel for drawing can eliminate this problem by providing a dedicated area for graphics.

Swing components that inherit from class JComponent contain method paintComponent that helps them draw properly in the context of a Swing GUI. When customizing a JPanel for use as a dedicated drawing area, the subclass should override method paintComponent and call the superclass version of paintComponent as the first statement in the body of the overridden method. This ensures that painting occurs in the proper order and that Swing’s painting mechanism remains intact. An important part of this mechanism is that subclasses of JComponent support transparency, which can be set with method setOpaque (a false argument indicates the component is transparent). To paint a component correctly, the program must determine whether the component is transparent. The code that performs this check is in the superclass version of paintComponent. When a component is transparent, paintComponent will not clear the component’s background when the program paints the component. When a component is opaque, paintComponent clears the background before continuing the painting operation. If the superclass version of paintComponent is not called, an opaque GUI component typically will not display correctly on the user interface. Also, if the superclass version is called after performing the customized drawing statements, the results typically will be erased.

Look-and-Feel Observation 13.4

When overriding a JComponent’s paintComponent method, the first statement in the body should always be a call to the superclass’s original version of the method.

Common Programming Error 13.1

When overriding a JComponent’s paintComponent method, not calling the superclass’s original version of paintComponent might prevent the GUI component from displaying properly on the GUI.

Common Programming Error 13.2

When overriding a JComponent’s paintComponent method, calling the superclass’s paintComponent method after other drawing is performed erases the other drawings.

Classes JFrame and JApplet are not subclasses of JComponent; therefore, they do not contain method paintComponent. To draw directly on subclasses of JFrame and JApplet, override method paint.

Look-and-Feel Observation 13.5

Calling repaint for a Swing GUI component indicates that the component should be painted as soon as possible. The background of the GUI component is cleared only if the component is opaque. Most Swing components are transparent by default. JComponent method setOpaque can be passed a boolean argument indicating whether the component is opaque (true) or transparent (false). The GUI components of package java.awt are different from Swing components, in that repaint results in a call to Component method update (which clears the component’s background) and update calls method paint (rather than paintComponent).
The program of Fig. 13.2 and Fig. 13.3 demonstrates a customized subclass of JPanel. Class CustomPanel (Fig. 13.2) has its own paintComponent method that draws a circle or a square, depending on the value passed to CustomPanel’s draw method. For this purpose, CustomPanel line 11 defines constants that enable the program to specify the shape a CustomPanel draws on itself with each call to its paintComponent method. Class CustomPanelTest (Fig. 13.3) creates a CustomPanel and a GUI that enable the user to choose which shape to draw.

Class CustomPanel contains one instance variable, shape, that stores an integer representing the shape to draw. Method paintComponent (lines 15–23) draws a shape on the panel. If shape is CIRCLE, Graphics method fillOval draws a solid circle. If shape is SQUARE, Graphics method fillRect draws a solid square. Method draw (lines 26–30) sets instance variable shape and calls repaint to refresh the CustomPanel object. Note that calling repaint (which is really this.repaint()) for the CustomPanel schedules a painting operation for the CustomPanel. Method paintComponent will be called to repaint the CustomPanel and draw the appropriate shape.

```java
public class CustomPanel extends JPanel {
    public final static int CIRCLE = 1, SQUARE = 2;
    private int shape;

    // use shape to draw an oval or rectangle
    public void paintComponent( Graphics g ) {
        super.paintComponent( g );
        if ( shape == CIRCLE )
            g.fillOval( 50, 10, 60, 60 );
        else if ( shape == SQUARE )
            g.fillRect( 50, 10, 60, 60 );
    }

    // set shape value and repaint CustomPanel
    public void draw( int shapeToDraw ) {
        shape = shapeToDraw;
        repaint();
    }
}
```

Fig. 13.2 Defining a custom drawing area by subclassing JPanel.
Class `CustomPanelTest` (Fig. 13.3) instantiates a `CustomPanel` object (line 22 of its constructor) and sets its background color to green, so the `CustomPanel` area is visible on the application. Next, the constructor instantiates `JButton` objects `squareButton` and `circleButton`. Lines 27–40 register an event handler for `squareButton`'s `ActionEvent`. Lines 43–56 register an event handler for `circleButton`'s `ActionEvent`. Lines 35 and 51 each call `CustomPanel` method `draw`. In each case, the appropriate constant (`CustomPanel.SQUARE` or `CustomPanel.CIRCLE`) is passed as an argument to indicate which shape to draw.
circleButton.addActionListener(
    // anonymous inner class to handle circleButton events
    new ActionListener() {
        // draw a circle
        public void actionPerformed( ActionEvent event )
        {
            myPanel.draw( CustomPanel.CIRCLE );
        }
    } // end anonymous inner class
); // end call to addActionListener

// set up panel containing buttons
buttonPanel = new JPanel();
buttonPanel.setLayout( new GridLayout( 1, 2 ) );
buttonPanel.add( circleButton );
buttonPanel.add( squareButton );

// attach button panel & custom drawing area to content pane
Container container = getContentPane();
container.add( myPanel, BorderLayout.CENTER );
container.add( buttonPanel, BorderLayout.SOUTH );

setSize( 300, 150 );
setVisible( true );

// execute application
public static void main( String args[] )
{
    CustomPanelTest application = new CustomPanelTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
} // end class CustomPanelTest

Fig. 13.3 Drawing on a customized subclass of class JPanel (part 2 of 2).

For layout of the buttons, CustomPanelTest creates JPanel buttonPanel with a GridLayout of one row and two columns (lines 59–60), then attaches the buttons to the panel (lines 61–62). Finally, CustomPanelTest adds myPanel to the CENTER
region of the content pane and adds buttonPanel to the SOUTH region of the content pane. Note that the BorderLayout expands myPanel to fill the center region.

13.4 Creating a Self-Contained Subclass of JPanel

JPanel s do not support conventional events supported by other GUI components, like buttons, text fields and windows. However, JPanel s are capable of recognizing such lower-level events as mouse events and key events. The program of Fig. 13.4 and Fig. 13.5 allows the user to draw an oval on a subclass of JPanel by dragging the mouse across the panel. Class SelfContainedPanel (Fig. 13.4) listens for its own mouse events and draws an oval on itself in response to those mouse events. The location and size of the oval are determined from the coordinates of the mouse events. The coordinates at which the user presses the mouse button specify the starting point for the oval’s bounding box. As the user drags the mouse, the coordinates of the mouse pointer specify another point. Together, the program uses these points to calculate the upper-left x-y coordinate, the width and the height of the oval’s bounding box. The size of the oval changes continuously while the user drags the mouse. When the user releases the mouse button, the program calculates the final bounding box for the oval and draws the oval. Line 4 of Fig. 13.4 indicates that class SelfContainedPanel is in package com.deitel.jhtp4.ch13 for future reuse. Class SelfContainedPanelTest imports SelfContainedPanel at line 13 of Fig. 13.5.

```java
// Fig. 13.4: SelfContainedPanel.java
// A self-contained JPanel class that handles its own mouse events.
package com.deitel.jhtp4.ch13;

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

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

public class SelfContainedPanel extends JPanel {
    private int x1, y1, x2, y2;

    // set up mouse event handling for SelfContainedPanel
    public SelfContainedPanel()
    {
        // set up mouse listener
        addMouseListener(

            // anonymous inner class for mouse pressed and released event handling
            new MouseAdapter() {

                // handle mouse press event
                public void mousePressed( MouseEvent event )
                {
                    x1 = event.getX();
                }
            }
        );
    }
}
```

Fig. 13.4 Customized subclass of JPanel that processes mouse events (part 1 of 2).
Fig. 13.4 Customized subclass of JPanel that processes mouse events (part 2 of 2).
Class `SelfContainedPanel` (Fig. 13.4) extends class `JPanel`. Instance variables `x1` and `y1` store the initial coordinates where the `mousePressed` event occurs on the `SelfContainedPanel`. Instance variables `x2` and `y2` store the coordinates where the user drags the mouse or releases the mouse button. All the coordinates are with respect to the upper-left corner of the `SelfContainedPanel`.

**Look-and-Feel Observation 13.6**

Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component.

The `SelfContainedPanel` constructor (lines 17–63) uses methods `addMouseListener` and `addMouseMotionListener` to register anonymous inner-class objects to handle mouse events and mouse motion events for the `SelfContainedPanel`. Only `mousePressed` (lines 27–31), `mouseReleased` (lines 34–39) and `mouseDragged` (lines 52–57) are overridden to perform tasks. The other mouse event-handling methods are inherited by the anonymous inner classes from the adapter classes `MouseListener` and `MouseMotionListener`.

By extending class `JPanel`, we are actually creating a new GUI component. Layout managers often use a GUI component’s `getPreferredSize` method (inherited from class `java.awt.Component`) to determine the preferred width and height of a component when laying out that component as part of a GUI. If a new component has a preferred width and height, it should override method `getPreferredSize` (lines 66–69) to return that width and height as an object of class `Dimension` (package `java.awt`).

**Look-and-Feel Observation 13.7**

The default size of a `JPanel` object is 10 pixels wide and 10 pixels tall.

**Look-and-Feel Observation 13.8**

When subclassing `JPanel` (or any other `JComponent`), override method `getPreferredSize` if the new component should have a specific preferred width and height.

Method `paintComponent` (lines 72–78) draws an oval, using the current values of instance variables `x1`, `y1`, `x2` and `y2`. The width, height and upper-left corner are determined by the pressing and holding of the mouse button, the dragging of the mouse and releasing of the mouse button on the `SelfContainedPanel` drawing area.

The initial coordinates `x1` and `y1` on the `SelfContainedPanel` drawing area are captured in method `mousePressed` (lines 27–31). As the user drags the mouse after the initial `mousePressed` operation, the program generates a series of calls to `mouseDragged` (lines 52–57) while the user continues to hold the mouse button and move the mouse. Each call captures in variables `x2` and `y2` the current location of the mouse with respect to the upper-left corner of the `SelfContainedPanel` and calls `repaint` to draw the current version of the oval. Drawing is strictly confined to the `SelfContainedPanel`, even if the user drags outside the `SelfContainedPanel` drawing area. Anything drawn off the `SelfContainedPanel` is clipped—pixels are not displayed outside the bounds of the `SelfContainedPanel`.

The calculations provided in method `paintComponent` determine the proper upper-left corner, using method `Math.min` twice to find the smaller `x` coordinate and `y` coordinate. The oval’s width and height must be positive values or the oval is not displayed.
Method `Math.abs` gets the absolute value of the subtractions \( x1 - x2 \) and \( y1 - y2 \) that determine the width and height of the oval’s bounding rectangle, respectively. When the calculations are complete, `paintComponent` draws the oval. The call to the superclass version of `paintComponent` at the beginning of the method guarantees that the previous oval displayed on the `SelfContainedPanel` is erased before the new one is displayed.

**Look-and-Feel Observation 13.9**

Most Swing GUI components can be transparent or opaque. If a Swing GUI component is opaque, when its `paintComponent` method is called, its background will be cleared. Otherwise, its background will not be cleared. Only opaque components can display a customized background color.

**Look-and-Feel Observation 13.10**

`JPanel` objects are opaque by default.

When the user releases the mouse button, method `mouseReleased` (lines 34–39) captures in variables \( x2 \) and \( y2 \) the final location of the mouse and invokes `repaint` to draw the final version of the oval.

Class `SelfContainedPanelTest`’s constructor (lines 21–57 of Fig. 13.5) creates an instance of class `SelfContainedPanel` (line 24) and sets the background color (line 25) of the `SelfContainedPanel` to yellow so that its area is visible against the background of the application window.

```java
public class SelfContainedPanelTest extends JFrame {

    private SelfContainedPanel myPanel;

    // set up GUI and mouse motion event handlers for application window
    public SelfContainedPanelTest() {
        // set up a SelfContainedPanel
        myPanel = new SelfContainedPanel();
        myPanel.setBackground( Color.yellow );
        Container container = getContentPane();

        // Fig. 13.5: SelfContainedPanelTest.java
        // Creating a self-contained subclass of JPanel
        // that processes its own mouse events.

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

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

        // Deitel packages
    }
}
```

Fig. 13.5 Capturing mouse events with a `JPanel` (part 1 of 3).
container.setLayout( new FlowLayout() );
container.add( myPanel );

// set up mouse motion event handling
addMouseMotionListener(
    new MouseMotionListener()
    {
        // anonymous inner class for mouse motion event handling
        public void mouseDragged( MouseEvent event )
        {
            setTitle( "Dragging: x=" + event.getX() + "; y=" + event.getY() );
        }

        // handle mouse move event
        public void mouseMoved( MouseEvent event )
        {
            setTitle( "Moving: x=" + event.getX() + "; y=" + event.getY() );
        }
    }  // end anonymous inner class
); // end call to addMouseMotionListener

setSize( 300, 200 );
setVisible( true );

// execute application
public static void main( String args[] )
{
    SelfContainedPanelTest application =
        new SelfContainedPanelTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
} // end class SelfContainedPanelTest

Fig. 13.5 Capturing mouse events with a JPanel (part 2 of 3).
We would like this program to distinguish between mouse motion events on the
SelfContainedPanel and mouse motion events on the application window, so lines
32–53 register an object of an anonymous inner class to handle the application’s mouse
motion events. Event handlers mouseDragged (lines 38–42) and mouseMoved (lines
45–49) use method setTitle (inherited from class java.awt.Frame) to display a
String in the window’s title bar indicating the x-y coordinate where the mouse motion
event occurred.

When executing this program, try dragging from the background of the application
window into the SelfContainedPanel area to see that the drag events are sent to the
application window rather than the SelfContainedPanel. Then, start a new drag
operation in the SelfContainedPanel area and drag out to the background of the
application window to see that the drag events are sent to the SelfContainedPanel
rather than to the application window.

Look-and-Feel Observation 13.11
A mouse drag operation begins with a mouse-pressed event. All subsequent mouse drag
events (until the user releases the mouse button) are sent to the GUI component that received
the original mouse-pressed event.

13.5 JSlider

JSliders enable the user to select from a range of integer values. Class JSlider inher-
its from JComponent. Figure 13.6 shows a horizontal JSlider with tick marks and the
thumb that allows the user to select a value. JSliders are highly customizable in that they
can display major tick marks, minor tick marks and labels for the tick marks. They also sup-
port snap-to ticks where positioning the thumb between two tick marks causes the thumb
to snap to the closest tick mark.
Most Swing GUI components support user interactions through the mouse and the keyboard. For example, if a JSlider has the focus (i.e., it is the currently selected GUI component in the user interface), the left arrow key and right arrow key cause the thumb of the JSlider to decrease or increase by 1, respectively. The down arrow key and up arrow key also cause the thumb of the JSlider to decrease or increase by 1, respectively. The PgDn key (page down) and PgUp key (page up) cause the thumb of the JSlider to decrease or increase by block increments of one-tenth of the range of values, respectively. The Home key moves the thumb to the minimum value of the JSlider and the End key moves the thumb to the maximum value of the JSlider.

**Look-and-Feel Observation 13.12**
Most Swing components support user interactions through the mouse and the keyboard.

JSliders have either a horizontal orientation or a vertical orientation. For a horizontal JSlider, the minimum value is at the extreme left and the maximum value is at the extreme right of the JSlider. For a vertical JSlider, the minimum value is at the extreme bottom and the maximum value is at the extreme top of the JSlider. The relative position of the thumb indicates the current value of the JSlider.

**Look-and-Feel Observation 13.13**
The minimum and maximum value positions on a JSlider can be switched by calling the JSlider method setInverted with boolean argument true.

The program of Fig. 13.7 and Fig. 13.8 allows the user to size a circle drawn on a subclass of JPanel called OvalPanel (Fig. 13.7). The user specifies the diameter of the circle with a horizontal JSlider. Application class SliderDemo (Fig. 13.8) creates the JSlider that controls the diameter of the circle. Class OvalPanel is a subclass of JPanel that knows how to draw a circle on itself, using its own instance variable diameter to determine the diameter of the circle—the diameter is used as the width and height of the bounding box in which the circle is displayed. The diameter value is set when the user interacts with the JSlider. The event handler calls method setDiameter in class OvalPanel to set the diameter and calls repaint to draw the new circle. The repaint call results in a call to OvalPanel’s paintComponent method.

Class OvalPanel (Fig. 13.7) contains a paintComponent method (lines 14–19) that draws a filled oval (a circle in this example), a setDiameter method (lines 22–28) that changes the diameter of the circle and repaints the OvalPanel, a getPreferredSize method (lines 31–34) that defines the preferred width and height of an OvalPanel and a getMinimumSize method (lines 37–40) that defines the minimum width and height of an OvalPanel.

**Look-and-Feel Observation 13.14**
If a new GUI component has a minimum width and height (i.e., smaller dimensions would render the component ineffective on the display), override method getMinimumSize to return the minimum width and height as an instance of class Dimension.

**Look-and-Feel Observation 13.15**
For many GUI components, method getMinimumSize is defined to return the result of a call to that component’s getPreferredSize method.
Class SliderDemo’s constructor (lines 17–54 of Fig. 13.8) instantiates OvalPanel object myPanel and sets its background color (lines 22–23). Lines 26–27 instantiate JSlider object diameterSlider to control the diameter of the circle drawn on the OvalPanel. The orientation of diameterSlider is HORIZONTAL (a constant in interface SwingConstants). The second and third constructor arguments to the JSlider constructor indicate the minimum and maximum integer values in the range of values for this JSlider. The last constructor argument indicates that the initial value of the JSlider (i.e., where the thumb is displayed) should be 10.
// Fig. 13.8: SliderDemo.java
// Using JSliders to size an oval.

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

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

public class SliderDemo extends JFrame {

    private JSlider diameterSlider;
    private OvalPanel myPanel;

    // set up GUI
    public SliderDemo() {
        super( "Slider Demo" );

        // set up OvalPanel
        myPanel = new OvalPanel();
        myPanel.setBackground( Color.yellow );

        // set up JSlider to control diameter value
        diameterSlider =
            new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
        diameterSlider.setMajorTickSpacing( 10 );
        diameterSlider.setPaintTicks( true );

        // register JSlider event listener
        diameterSlider.addChangeListener(
            new ChangeListener() {
                // handle change in slider value
                public void stateChanged( ChangeEvent e ) {
                    myPanel.setDiameter( diameterSlider.getValue() );
                }
            }  // end anonymous inner class
        ); // end call to addChangeListener

        // attach components to content pane
        Container container = getContentPane();
        container.add( diameterSlider, BorderLayout.SOUTH );
        container.add( myPanel, BorderLayout.CENTER );

        setSize( 220, 270 );
    }

Fig. 13.8 Using a JSlider to determine the diameter of a circle (part 1 of 2).
Chapter 13
Graphical User Interface Components: Part 2

Line 28–29 customize the appearance of the `JSlider`. Method `setMajorTickSpacing` indicates that each tick mark represents 10 values in the range of values supported by the `JSlider`. Method `setPaintTicks` with a `true` argument indicates that the tick marks should be displayed (they are not displayed by default). See the `JSlider` on-line documentation for more information on methods that are used to customize a `JSlider`’s appearance.

`JSliders` generate `ChangeEvent`s (package `javax.swing.event`) when the user interacts with a `JSlider`. An object of a class that implements interface `ChangeListener` (package `javax.swing.event`) and defines method `stateChanged` can respond to `ChangeEvent`s. Lines 32–45 register an object of an anonymous inner class that implements `ChangeListener` to handle `diameterSlider`’s events. When method `stateChanged` is called in response to a user interaction, it calls `myPanel`'s `setDiameter` method and passes the current value of the `JSlider` as an argument. Method `getValue` of class `JSlider` returns the current thumb position.

### 13.6 Windows

From Chapter 9 to this chapter, most applications have used an instance of a subclass of `JFrame` as the application window. In this section, we discuss several important issues regarding `JFrames`.  

```java
53     setVisible( true );
54 }
55 // execute application
56 public static void main( String args[] )
57 {
58     SliderDemo application = new SliderDemo();
59     application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
60 }
61 // end class SliderDemo
```

**Fig. 13.8** Using a `JSlider` to determine the diameter of a circle (part 2 of 2).
A **JFrame** is a window with a title bar and a border. Class **JFrame** is a subclass of **java.awt.Frame** (which is a subclass of **java.awt.Window**). As such, **JFrame** is one of the few Swing GUI components that is not a lightweight GUI component. Unlike most Swing components, **JFrame** is not written completely in Java. In fact, when you display a window from a Java program, the window is provided by the local platform’s set of GUI components—the window will look like all other windows displayed on that platform. When a Java program executes on a Macintosh and displays a window, the window’s title bar and borders will look like other Macintosh applications. When a Java program executes on Microsoft Windows and displays a window, the window’s title bar and borders will look like other Microsoft Windows applications. And when a Java program executes on a Unix platform and displays a window, the window’s title bar and borders will look like other Unix applications on that platform.

Class **JFrame** supports three operations when the user closes the window. By default, a window is hidden (i.e., removed from the screen) when the user closes a window. This can be controlled with **JFrame** method **setDefaultCloseOperation**. Interface **WindowConstants** (package **javax.swing**) defines three constants for use with this method—**DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE** and **HIDE_ON_CLOSE** (the default). Most platforms only allow a limited number of windows to be displayed on the screen. As such, a window is a valuable resource that should be given back to the system when it is no longer needed. Class **Window** (an indirect superclass of **JFrame**) defines method **dispose** for this purpose. When a **Window** is no longer needed in an application, you should explicitly **dispose** of the **Window**. This can be done by calling the **Window**’s **dispose** method or by calling method **setDefaultCloseOperation** with the argument **WindowConstants.DISPOSE_ON_CLOSE**. Also, terminating an application will return window resources to the system. Setting the default close operation to **DO NOTHING ON CLOSE** indicates that you will determine what to do when the user indicates that the window should be closed.

**Software Engineering Observation 13.1**

Windows are a valuable system resource that should be returned to the system when they are no longer needed.

By default, a window is not displayed on the screen until the program invokes the window’s **setVisible** method (inherited from class **java.awt.Component**) with a **true** argument or invokes the window’s **show** method, which takes no arguments. Also, a window’s size should be set with a call to method **setSize** (inherited from class **java.awt.Component**). The position of a window when it appears on the screen is specified with method **setLocation** (inherited from class **java.awt.Component**).

**Common Programming Error 13.3**

Forgetting to call method **show** or method **setVisible** on a window is a run-time logic error; the window is not displayed.

**Common Programming Error 13.4**

Forgetting to call the **setSize** method on a window is a run-time logic error—only the title bar appears.

All windows generate window events when the user manipulates the window. Event listeners are registered for window events with method **addWindowListener** of class.
**Window.** Interface **WindowListener** (implemented by window event listeners) provides seven methods for handling window events—`windowActivated` (called when the user makes a window the active window), `windowClosed` (called after the window is closed), `windowClosing` (called when the user initiates closing of the window), `windowDeactivated` (called when the user makes another window the active window), `windowIconified` (called when the user minimizes a window), `windowDeiconified` (called when the user restores a window from being minimized) and `windowOpened` (called when a program first displays a window on the screen).

Most windows have an icon at the top-left or top-right corner that enables a user to close the window and terminate a program. Most windows also have an icon in the upper-left corner of the window that displays a menu when the user clicks the icon. This menu normally contains a **Close** option to close the window and several other options for manipulating the window.

### 13.7 Designing Programs that Execute as Applets or Applications

It is sometimes desirable to design a Java program that can execute both as a stand-alone application and as an applet in a Web browser. Such a program can be used to provide the same functionality to users worldwide by making the applet available for download via Web and can be installed on a computer as a stand-alone application. The next example discusses how to create a small program that can execute both as an applet and as an application. [Note: In Chapter 16 and Chapter 17, we discuss several issues that make applets different from applications and security restrictions placed on applets that prevent certain application features from working in an applet.]

Frequently, programs use **JFrames** to create **GUI-based applications**. The **JFrame** provides the space in which the application GUI appears. When the user closes the **JFrame**, the application terminates. In this section, we demonstrate how to convert an applet into a GUI-based application. The program of Fig. 13.9 presents an applet that also can be executed as an application.

**Software Engineering Observation 13.2**

> When designing a program to execute as both an applet and an application, begin by defining it as an applet, because applets have limitations due to security restrictions imposed on them by Web browsers. If the program executes properly as an applet, it can be made to work properly as an application. However, the reverse is not always true.

Our applet class **DrawShapes** presents the user with three buttons which, when pressed, cause an instance of class **DrawPanel** (lines 121–126) to draw a random line, rectangle or oval (depending on which button is pressed). The applet does not contain any new features as far as GUI components, layouts or drawing are concerned. The only new feature is that the **DrawShapes** class now also contains a **main** method (lines 60–99) that can be used to execute the program as an application. We discuss this method in detail below.

The HTML document that loads the applet into the **appletviewer** or a Web browser specifies the applet’s width and height as 300 and 200, respectively. When the program executes as an application with the **java** interpreter, you can supply arguments to the program (called **command-line arguments**) that specify the width and height of the application window. For example, the command
contains two command-line arguments—600 and 400—that specify the width and height of the application window. Java passes the command-line arguments to `main` as the array of `Strings` called `args`, which we have declared in the parameter list of every application’s `main` method, but not used until this point. The first argument after the application class name is the first `String` in the array `args`, and the length of the array is the total number of command-line arguments. Line 60 begins the definition of `main` and declares array `args` as an array of `Strings` that allows the application to access the command-line arguments. Line 62 defines variables `width` and `height` that are used to specify the size of the application window.

```java
// Fig. 13.9: DrawShapes.java
// Draw random lines, rectangles and ovals

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

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

public class DrawShapes extends JApplet {

    private JButton choices[];
    private String names[] = { "Line", "Rectangle", "Oval" };
    private JPanel buttonPanel;
    private DrawPanel drawingPanel;
    private int width = 300, height = 200;

    // initialize applet; set up GUI
    public void init() {
        // set up DrawPanel
drawingPanel = new DrawPanel( width, height );

        // create array of buttons
choices = new JButton[ names.length ];

        // set up panel for buttons
buttonPanel = new JPanel();
        buttonPanel.setLayout( new GridLayout( 1, choices.length ) );

        // set up buttons and register their listeners
        ButtonHandler handler = new ButtonHandler();
        for ( int count = 0; count < choices.length; count++ ) {
            choices[ count ] = new JButton( names[ count ] );
            buttonPanel.add( choices[ count ] );
            choices[ count ].addActionListener( handler );
        }
    }

    // draw random shapes
    public void paint(Graphics g) {
        // code to draw shapes
    }

    public static void main(String[] args) {
        // code to run applet
    }
}
```

Fig. 13.9 Creating a GUI-based application from an applet (part 1 of 4).
// attach components to content pane
Container container = getContentPane();
container.add( buttonPanel, BorderLayout.NORTH );
container.add( drawingPanel, BorderLayout.CENTER );

// enables application to specify width of drawing area
public void setWidth( int newWidth )
{
    width = ( newWidth >= 0 ? newWidth : 300 );
}

// enables application to specify height of drawing area
public void setHeight( int newHeight )
{
    height = ( newHeight >= 0 ? newHeight : 200 );
}

// execute applet as an application
public static void main( String args[] )
{
    int width, height;

    // check for command-line arguments
    if ( args.length != 2 ) {
        width = 300;
        height = 200;
    }
    else {
        width = Integer.parseInt( args[ 0 ] );
        height = Integer.parseInt( args[ 1 ] );
    }

    // create window in which applet will execute
    JFrame applicationWindow =
        new JFrame( "An applet running as an application" );

    applicationWindow.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE );

    // create one applet instance
    DrawShapes appletObject = new DrawShapes();
    appletObject.setWidth( width );
    appletObject.setHeight( height );

    // call applet's init and start methods
    appletObject.init();
    appletObject.start();

    // attach applet to center of window
    applicationWindow.getContentPane().add( appletObject );
// set the window's size
applicationWindow.setSize( width, height );

// showing the window causes all GUI components
// attached to the window to be painted
applicationWindow.setVisible( true );

// private inner class to handle button events
private class ButtonHandler implements ActionListener {
    // determine button user pressed and set drawing area's
    // current choice
    public void actionPerformed( ActionEvent event )
    {
        for ( int count = 0; count < choices.length; count++ )
        {
            if ( event.getSource() == choices[ count ] ) {
                drawingPanel.setCurrentChoice( count );
                break;
            }
        }
    }
}

} // end private inner class ButtonHandler

} // end class DrawShapes

// subclass of JPanel to allow drawing in a separate area
class DrawPanel extends JPanel {
    private int currentChoice = -1;  // don't draw first time
    private int width = 100, height = 100;
    
    // initialize width and height of DrawPanel
    public DrawPanel( int newWidth, int newHeight )
    {
        width = ( newWidth >= 0 ? newWidth : 100 );
        height = ( newHeight >= 0 ? newHeight : 100 );
    }
    
    // draw line, rectangle or oval based on user's choice
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        switch( currentChoice ) {
        case 0:
            g.drawLine( randomX(), randomY(),
                        randomX(), randomY() );
            break;
        }
Lines 65–72 determine the initial width and height of the application window. The if condition determines the length of array args. If the number of elements is not 2, the width and height are set to 300 and 200 by default. Otherwise, lines 70–71 convert the command-line arguments from Strings to int values with parseInt and use them as the width...
and \texttt{height}. [Note: This program assumes that the user inputs whole-number values for the command-line arguments; if not, an exception will occur. In Chapter 14, we discuss how to make our programs more robust by dealing with improper values when they occur.]

When an applet executes, the window in which it executes is supplied by the applet container (i.e., the \texttt{appletviewer} or browser). When a program executes as an application, the application must create its own window (if one is required). Lines 75–76 create the \texttt{JFrame} to which the program will attach the applet. As with any \texttt{JFrame} that is used as the application’s primary window, you should provide a mechanism to terminate the application. Lines 78–79 specify that the application should terminate when the user closes the window.

\begin{center}
Software Engineering Observation 13.3
\end{center}

\textit{To execute an applet as an application, the application must provide a window in which the applet can be displayed.}

When an applet executes in an applet container, the container creates one object of the applet class to execute the applet’s tasks. In an application, objects are not created unless the application explicitly contains statements that create objects. Line 82 defines one instance of applet class \texttt{DrawShapes}. Notice the call to the no-argument constructor. We did not define a constructor in class \texttt{DrawShapes} (applet classes typically do not define constructors). Remember that the compiler provides a default constructor for a class that does not define any constructors. Lines 83–84 call \texttt{DrawShapes} methods \texttt{setWidth} and \texttt{setHeight} to validate the values for \texttt{width} and \texttt{height} (improper values are set to 300 and 200, respectively).

\begin{center}
Software Engineering Observation 13.4
\end{center}

\textit{To execute an applet as an application, the application must create an instance of the applet class to execute.}

When an applet executes in an applet container, the container guarantees that methods \texttt{init}, \texttt{start} and \texttt{paint} will be called to begin the applet’s execution. However, these methods are not special to an application. Method \texttt{init} and \texttt{start} are not invoked automatically or required by an application. (Method \texttt{paint} is part of any window and will be called when it is necessary to repaint the window.) Lines 87–88 invoke \texttt{appletObject}’s \texttt{init} method to initialize the applet and set up its GUI, then invoke method \texttt{start}. [Note: In our example, we did not override method \texttt{start}. It is called here to mimic the start-up sequence normally followed for an applet.]

\begin{center}
Software Engineering Observation 13.5
\end{center}

\textit{When executing an applet as an application, the application must call \texttt{init} and \texttt{start} explicitly to simulate the normal applet start-up sequence of method calls.}

When an applet executes in an applet container, the applet is normally attached to the applet container’s window. An application must explicitly attach an applet object to the application window. Line 91 obtains a reference to the \texttt{applicationWindow}’s content pane and adds the \texttt{appletObject} to the default \texttt{CENTER} of the content pane’s \texttt{BorderLayout}. The \texttt{appletObject} will occupy the entire window.

\begin{center}
Software Engineering Observation 13.6
\end{center}

\textit{When one is executing an applet as an application, the application must attach the applet object to its window.}
Finally, the application window must be sized and displayed on the screen. Line 94 sets the application window’s size, and line 98 displays the window. When a Java program displays any window, all the components attached to the window receive calls to their `paint` methods (if they are heavyweight components) or their `paintComponent` methods (if they are lightweight components). Thus, displaying the application window results in a call to the applet’s `paint` method to complete the normal start-up sequence for the applet.

Try executing this program as an applet and as an application to see that it has the same functionality when executed.

### 13.8 Using Menus with Frames

*Menus* are an integral part of GUIs. Menus allow the user to perform actions without unnecessarily “cluttering” a graphical user interface with extra GUI components. In Swing GUIs, menus can be attached only to objects of the classes that provide method `setJMenuBar`. Two such classes are `JFrame` and `JApplet`. The classes used to define menus are `JMenuBar`, `JMenuItem`, `JMenu`, `JCheckBoxMenuItem` and class `JRadioButtonMenuItem`.

**Look-and-Feel Observation 13.16**

Menus simplify GUIs by reducing the number of components the user views.

Class `JMenuBar` (a subclass of `JComponent`) contains the methods necessary to manage a *menu bar*, which is a container for menus.

Class `JMenuItem` (a subclass of `javax.swing.AbstractButton`) contains the methods necessary to manage *menu items*. A menu item is a GUI component inside a menu that, when selected, causes an action to be performed. A menu item can be used to initiate an action or it can be a *submenu* that provides more menu items from which the user can select. Submenus are useful for grouping related menu items in a menu.

Class `JMenu` (a subclass of `javax.swing.JMenuItem`) contains the methods necessary for managing *menus*. Menus contain menu items and are added to menu bars or to other menus as submenus. When a menu is clicked, the menu expands to show its list of menu items. Clicking a menu item generates an action event.

Class `JCheckBoxMenuItem` (a subclass of `javax.swing.JMenuItem`) contains the methods necessary to manage menu items that can be toggled on or off. When a `JCheckBoxMenuItem` is selected, a check appears to the left of the menu item. When the `JCheckBoxMenuItem` is selected again, the check to the left of the previously selected menu item is removed.

Class `JRadioButtonMenuItem` (a subclass of `javax.swing.JMenuItem`) contains the methods necessary to manage menu items that can be toggled on or off like `JCheckBoxMenuItem`. When multiple `JRadioButtonMenuItem` are maintained as part of a `ButtonGroup`, only one item in the group can be selected at a given time. When a `JRadioButtonMenuItem` is selected, a filled circle appears to the left of the menu item. When another `JRadioButtonMenuItem` is selected, the filled circle to the left of the previously selected menu item is removed.

The application of Fig. 13.10 demonstrates various types of menu items. The program also demonstrates how to specify special characters called mnemonics that can provide quick access to a menu or menu item from the keyboard. Mnemonics can be used with objects of all classes that have subclass `javax.swing.AbstractButton`. 


Figure 13.10: MenuTest.java

Demonstrating menus

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

Java extension packages
import javax.swing.*;

public class MenuTest extends JFrame {
  private Color colorValues[] =
    { Color.black, Color.blue, Color.red, Color.green };

  private JRadioButtonMenuItem colorItems[], fonts[];
  private JCheckBoxMenuItem styleItems[];
  private JLabel displayLabel;
  private ButtonGroup fontGroup, colorGroup;
  private int style;

  // set up GUI
  public MenuTest() {
    super( "Using JMenus" );

    // set up File menu and its menu items
    JMenu fileMenu = new JMenu( "File" );
    fileMenu.setMnemonic( 'F' );

    // set up About... menu item
    JMenuItem aboutItem = new JMenuItem( "About..." );
    aboutItem.setMnemonic( 'A' );

    aboutItem.addActionListener(
      new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
          JOptionPane.showMessageDialog( MenuTest.this, "This is an example
                                                  of using menus", "About", JOptionPane.PLAIN_MESSAGE );
        }
      }  // end anonymous inner class
    ); // end call to addActionListener

    fileMenu.add( aboutItem );
  }
}  // end anonymous inner class
// set up Exit menu item
JMenuItem exitItem = new JMenuItem( "Exit" );
exitItem.setMnemonic( 'x' );
exitItem.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            System.exit( 0 );
        }
    } // end anonymous inner class
); // end call to addActionListener

fileMenu.add( exitItem );

// create menu bar and attach it to MenuTest window
JMenuBar bar = new JMenuBar();
setJMenuBar( bar );
bar.add( fileMenu );

// create Format menu, its submenus and menu items
JMenu formatMenu = new JMenu( "Format" );
formatMenu.setMnemonic( 'r' );

// create Color submenu
String colors[] = { "Black", "Blue", "Red", "Green" };

JMenu colorMenu = new JMenu( "Color" );
colorMenu.setMnemonic( 'C' );

JRadioButtonMenuItem[] colorItems = new JRadioButtonMenuItem[ colors.length ];
colorGroup = new ButtonGroup();
ItemHandler itemHandler = new ItemHandler();

// create color radio button menu items
for ( int count = 0; count < colors.length; count++ ) {
    colorItems[ count ] =
        new JRadioButtonMenuItem( colors[ count ] );
    colorMenu.add( colorItems[ count ] );
    colorGroup.add( colorItems[ count ] );
    colorItems[ count ].addActionListener( itemHandler );
}

// select first Color menu item
colorItems[ 0 ]. setSelected( true );

Fig. 13.10 Using JMenus and mnemonics (part 2 of 5).
// add format menu to menu bar
formatMenu.add( colorMenu );
formatMenu.addSeparator();

// create Font submenu
String fontNames[] = { "Serif", "Monospaced", "SansSerif" };

JMenu fontMenu = new JMenu( "Font" );
fontMenu.setMnemonic( 'n' );

fonts = new JRadioButtonMenuItem[ fontNames.length ];
fontGroup = new ButtonGroup();

// create Font radio button menu items
for ( int count = 0; count < fonts.length; count++ ) {
    fonts[ count ] =
        new JRadioButtonMenuItem( fontNames[ count ] );
    fontMenu.add( fonts[ count ] );
    fontGroup.add( fonts[ count ] );
    fonts[ count ].addActionListener( itemHandler );
}

// select first Font menu item
fonts[ 0 ].setSelected( true );

fontMenu.addSeparator();

// set up style menu items
String styleNames[] = { "Bold", "Italic" };

styleItems = new JCheckBoxMenuItem[ styleNames.length ];
StyleHandler styleHandler = new StyleHandler();

// create style checkbox menu items
for ( int count = 0; count < styleNames.length; count++ ) {
    styleItems[ count ] =
        new JCheckBoxMenuItem( styleNames[ count ] );
    fontMenu.add( styleItems[ count ] );
    styleItems[ count ].addItemListener( styleHandler );
}

// put Font menu in Format menu
formatMenu.add( fontMenu );

// add Format menu to menu bar
bar.add( formatMenu );

Fig. 13.10 Using JMenu's and mnemonics (part 3 of 5).
// set up label to display text
displayLabel = new JLabel("Sample Text", SwingConstants.CENTER);
displayLabel.setForeground(colorValues[0]);
displayLabel.setFont(new Font("TimesRoman", Font.PLAIN, 72));

getContentPane().setBackground(Color.cyan);
getContentPane().add(displayLabel, BorderLayout.CENTER);

setSize(500, 200);
setVisible(true);

} // end constructor

// execute application
public static void main(String args[])
{
MenuTest application = new MenuTest();

application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

// inner class to handle action events from menu items
private class ItemHandler implements ActionListener {

// process color and font selections
public void actionPerformed(ActionEvent event)
{
    // process color selection
    for (int count = 0; count < colorItems.length; count++)
    {
        if (colorItems[count].isSelected())
        {
            displayLabel.setForeground(colorValues[count]);
            break;
        }
    }

    // process font selection
    for (int count = 0; count < fonts.length; count++)
    {
        if (event.getSource() == fonts[count])
        {
            displayLabel.setFont(new Font(fonts[count].getText(), style, 72));
            break;
        }
    }

    repaint();
}

} // end class ItemHandler

Fig. 13.10 Using JMenu and mnemonics (part 4 of 5).
Class `MenuTest` (line 11) is a completely self-contained class—it defines all the GUI components and event handling for the menu items. Most of the code for this application appears in the class’s constructor (lines 22–171).
Lines 27–72 set up the **File** menu and attach it to the menu bar. The **File** menu contains an **About...** menu item that displays a message dialog when the menu item is selected and an **Exit** menu item that can be selected to terminate the application.

Line 27 creates `fileMenu` and passes to the constructor the string “**File**” as the name of the menu. Line 28 uses `AbstractButton` method `setMnemonic` (inherited into class `JMenu`) to indicate that `F` is the mnemonic for this menu. Pressing the `Alt` key and the letter `F` opens the menu, just as clicking the menu name with the mouse would. In the GUI, the mnemonic character in the menu’s name is displayed with an underline (see the screen captures).

**Look-and-Feel Observation 13.17**

Mnemonics provide quick access to menu commands and button commands through the keyboard.

**Look-and-Feel Observation 13.18**

Different mnemonics should be used for each button or menu item. Normally, the first letter in the label on the menu item or button is used as the mnemonic. If multiple buttons or menu items start with the same letter, choose the next most prominent letter in the name (e.g., `x` is commonly chosen for a button or menu item called **Exit**).

Lines 31–32 define `JMenuItem aboutItem` with the name “**About...**” and set its mnemonic to the letter `A`. This menu item is added to `fileMenu` at line 51. To access the **About...** item through the keyboard, press the `Alt` key and letter `F` to open the **File** menu, then press `A` to select the **About...** menu item. Lines 34–49 create an `ActionListener` to process `aboutItem`’s action event. Lines 42–44 display a message dialog box. In most prior uses of `showMessageDialog`, the first argument has been `null`. The purpose of the first argument is to specify the *parent window* for the dialog box. The parent window helps determine where the dialog box will be displayed. If the parent window is specified as `null`, the dialog box appears in the center of the screen. If the parent window is not `null`, the dialog box appears centered over the specified parent window. In this example, the program specifies the parent window with `MenuTest.this`—the `this` reference of class `MenuTest`. When using the `this` reference in an inner class, specifying `this` by itself refers to the inner-class object. To reference the outer-class object’s `this` reference, qualify `this` with the outer-class name and a dot operator (.).

Dialog boxes can be either modal or modeless. A *modal dialog box* does not allow any other window in the application to be accessed until the dialog box is dismissed. A *modeless dialog box* allows other windows to be accessed while the dialog is displayed. By default, the dialogs displayed with class `JOptionPane` are modal dialogs. Class `JDialog` can be used to create your own modeless or modal dialogs.

Lines 54–72 define menu item `exitItem`, set its mnemonic to `x`, register an `ActionListener` that terminates the application when the user selects `exitItem` and add `exitItem` to the `fileMenu`.

Lines 75–77 create the `JMenuBar`, attach it to the application window with `JFrame` method `setJMenuBar` and use `JMenuBar` method `add` to attach the `fileMenu` to the menu bar.

**Common Programming Error 13.5**

Forgetting to set the menu bar with `JFrame` method `setJMenuBar` results in the menu bar not being displayed on the `JFrame`. 
Look-and-Feel Observation 13.19

Menus normally appear left to right in the order that they are added to a JMenuBar.

Lines 80–81 create menu formatMenu and set its mnemonic to r (F is not used because that is the File menu’s mnemonic).

Lines 86–87 create menu colorMenu (this will be a submenu in the Format menu) and set its mnemonic to C. Line 89 creates JRadioButtonMenuItem array colorItems that refers to the menu items in colorMenu. Line 90 creates the ButtonGroup colorGroup, which ensures that only one of the menu items in the Color submenu is selected at a time. Line 91 defines an instance of inner class ItemHandler (defined at lines 183–208) that responds to selections from the Color submenu and the Font submenu (discussed shortly). The for structure at lines 94–102 creates each JRadioButtonMenuItem in array colorItems, adds each menu item to colorMenu, adds each menu item to colorGroup and registers the ActionListener for each menu item.

Line 105 uses AbstractButton method setSelected to select the first element in the colorItems array. Line 108 adds the colorMenu as a submenu of the formatMenu.

Look-and-Feel Observation 13.20

Adding a menu as a menu item in another menu automatically makes the added menu a submenu. When the mouse is positioned over a submenu (or the submenu’s mnemonic is pressed), the submenu expands to show its menu items.

Line 109 adds a separator line to the menu. The separator appears as a horizontal line in the menu.

Look-and-Feel Observation 13.21

Separators can be added to a menu to group menu items logically.

Look-and-Feel Observation 13.22

Any lightweight GUI component (i.e., a component that subclasses JComponent) can be added to a JMenu or to a JMenuBar.

Lines 114–132 create the Font submenu and several JRadioButtonMenuItems and select the first element of JRadioButtonMenuItem array fonts. Line 139 creates a JCheckBoxMenuItem array to represent the menu items for specifying bold and italic styles for the fonts. Line 140 defines an instance of inner class StyleHandler (defined at lines 211–232) to respond to the JCheckBoxMenuItem events. The for structure at lines 143–150 creates each JCheckBoxMenuItem, adds each menu item to fontMenu and registers the ItemListener for each menu item. Line 153 adds fontMenu as a submenu of formatMenu. Line 156 adds the formatMenu to bar.

Lines 159–163 create a JLabel for which the Format menu items control the font, font color and font style. The initial foreground color is set to the first element of array colorValues (Color.black) and the initial font is set to TimesRoman with PLAIN style and 72-point size. Line 165 sets the background color of the window’s content pane to Color.cyan, and line 166 attaches the JLabel to the CENTER of the content pane’s BorderLayout.
Method `actionPerformed` of class `ItemHandler` (lines 186–206) uses two for structures to determine which font or color menu item generated the event and sets the font or color of the `JLabel display`, respectively. The if condition at line 191 uses `AbstractButton` method `isSelected` to determine the selected `JRadioButtonMenuItem`. The if condition at line 199 uses `EventSource` method `getSource` to get a reference to the `JRadioButtonMenuItem` that generated the event. Line 201 uses `AbstractButton` method `getText` to obtain the name of the font from the menu item.

The program calls method `itemStateChanged` of class `StyleHandler` (lines 214–230) if the user selects a `JCheckBoxMenuItem` in the `fontMenu`. Lines 219 and 223 determine whether either or both of the `JCheckBoxMenuItem`s are selected and use their combined state to determine the new style of the font.

### 13.9 Using JPopupMenu

Many of today’s computer applications provide so-called context-sensitive popup menus. In Swing, such menus are created with class `JPopupMenu` (a subclass of `JComponent`). These menus provide options that are specific to the component for which the popup trigger event was generated. On most systems, the popup trigger event occurs when the user presses and releases the right mouse button.

**Look-and-Feel Observation 13.23**

The popup trigger event is platform specific. On most platforms that use a mouse with multiple mouse buttons, the popup trigger event occurs when the user clicks the right mouse button.

Figure 13.11 creates a `JPopupMenu` that allows the user to select one of three colors and change the background color of the window. When the user clicks the right mouse button on the `PopupTest` window’s background, a `JPopupMenu` containing colors appears. If the user clicks one of the `JRadioButtonMenuItem`s that represents a color, method `actionPerformed` of class `ItemHandler` changes the background color of the window’s content pane.

```java
// Fig. 13.11: PopupTest.java
// Demonstrating JPopupMenu

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

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

// Fig. 13.11: Using a JPopupMenu object (part 1 of 3).
```
public PopupTest()
{
    super( "Using JPopupMenu" );

    ItemHandler handler = new ItemHandler();
    String colors[] = { "Blue", "Yellow", "Red" };

    // set up popup menu and its items
    ButtonGroup colorGroup = new ButtonGroup();
    popupMenu = new JPopupMenu();
    items = new JRadioButtonMenuItem[3];

    // construct each menu item and add to popup menu; also
    // enable event handling for each menu item
    for ( int count = 0; count < items.length; count++ ) {
        items[ count ] =
            new JRadioButtonMenuItem( colors[ count ] );

        popupMenu.add( items[ count ] );
        colorGroup.add( items[ count ] );

        items[ count ].addActionListener( handler );
    }

    get等内容。
Chapter 13  Graphical User Interface Components: Part 2  757

The constructor for class `PopupTest` (lines 20–79) defines the `JPopupMenu` at line 29. The `for` structure at lines 34–42 creates `JRadioButtonMenuItem`s to add to the `JPopupMenu`, adds them to the `JPopupMenu` (line 38), adds them to `ButtonGroup`
colorGroup (to maintain one selected JRadioButtonMenuItem at a time) and registers an ActionListener for each menu item.

Lines 48–75 register an instance of an anonymous inner class that extends MouseAdapter to handle the mouse events of the application window. Methods mousePressed (lines 54–57) and mouseReleased (lines 60–63) check for the popup-trigger event. Each method calls private utility method checkForTriggerEvent (lines 66–71) to determine whether the popup-trigger event occurred. MouseEvent method isPopupTrigger returns true if the popup-trigger event occurred. If so, method show of class JPopupMenu displays the JPopupMenu. The first argument to method show specifies the origin component, whose position helps determine where the JPopupMenu will appear on the screen. The last two arguments are the x-y coordinate from the origin component’s upper-left corner at which the JPopupMenu should appear.

Look-and-Feel Observation 13.24

Displaying a JPopupMenu for the popup-trigger event of multiple different GUI components requires registering mouse event handlers to check for the popup-trigger event for each of those GUI components.

When the user selects a menu item from the popup menu, class ItemHandler’s (lines 91–106) method actionPerformed (lines 94–104) determines which JRadioButtonMenuItem the user selected, then sets the background color of the window’s content pane.

13.10 Pluggable Look-and-Feel

A program that uses Java’s Abstract Windowing Toolkit GUI components (package java.awt) takes on the look-and-feel of the platform on which the program executes. A Java program running on a Macintosh looks like other programs running on a Macintosh. A Java program running on Microsoft Windows looks like other programs running on Microsoft Windows. A Java program running on a UNIX platform looks like other programs running on that UNIX platform. This could be desirable, because it allows users of the program on each platform to use the GUI components with which they are already familiar. However, this also introduces interesting portability issues.

Portability Tip 13.1

Programs that use Java’s Abstract Windowing Toolkit GUI components (package java.awt) take on the look-and-feel of the platform on which they execute.

Portability Tip 13.2

GUI components on each platform have different looks that can require different amounts of space to display. This could change the layout and alignments of GUI components.

Portability Tip 13.3

GUI components on each platform have different default functionality (e.g., some platforms allow a button with the focus to be “pressed” with the space bar, and some do not).

Swing’s lightweight GUI components eliminate many of these issues by providing uniform functionality across platforms and by defining a uniform cross-platform look-and-feel (known as the metal look-and-feel). Swing also provides the flexibility to customize
the look-and-feel to appear as a Microsoft Windows-style look-and-feel or a Motif-style (UNIX) look-and-feel.

The program of Fig. 13.12 demonstrates how to change the look-and-feel of a Swing GUI. The program creates several GUI components so you can see the change in the look-and-feel of several GUI components at the same time. The first output window shows the standard metal look-and-feel, the second output window shows the Motif look-and-feel, and the third output window shows the Windows look-and-feel.

```java
// Fig. 13.12: LookAndFeelDemo.java
// Changing the look and feel.

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

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

public class LookAndFeelDemo extends JFrame {

    private String strings[] = { "Metal", "Motif", "Windows" };
    private UIManager.LookAndFeelInfo looks[];
    private JRadioButton radio[];
    private ButtonGroup group;
    private JButton button;
    private JLabel label;
    private JComboBox comboBox;

    // set up GUI
    public LookAndFeelDemo()
    {
        super( "Look and Feel Demo" );
        Container container = getContentPane();
        // set up panel for NORTH of BorderLayout
        JPanel northPanel = new JPanel();
        northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );
        // set up label for NORTH panel
        label = new JLabel( "This is a Metal look-and-feel",
                            SwingConstants.CENTER );
        northPanel.add( label );
        // set up button for NORTH panel
        button = new JButton( "JButton" );
        northPanel.add( button );
        // set up combo box for NORTH panel
        comboBox = new JComboBox( strings );
        northPanel.add( comboBox );
    }
}
```

Fig. 13.12 Changing the look-and-feel of a Swing-based GUI (part 1 of 3).
// attach NORTH panel to content pane
carriage.add( northPanel, BorderLayout.NORTH );

// create array for radio buttons
radio = new JRadioButton[ strings.length ];

// set up panel for SOUTH of BorderLayout
JPanel southPanel = new JPanel();
        southPanel.setLayout(
                new GridLayout( 1, radio.length ) );

// set up radio buttons for SOUTH panel
        group = new ButtonGroup();
        ItemHandler handler = new ItemHandler();

for ( int count = 0; count < radio.length; count++ ) {
        radio[ count ] = new JRadioButton( strings[ count ] );
        radio[ count ].addItemListener( handler );
        group.add( radio[ count ] );
        southPanel.add( radio[ count ] );
    }

// attach SOUTH panel to content pane
container.add( southPanel, BorderLayout.SOUTH );

// get installed look-and-feel information
looks = UIManager.getInstalledLookAndFeels();

setSize( 300, 200 );
setVisible( true );

radio[ 0 ].setSelected( true );

// use UIManager to change look-and-feel of GUI
private void changeTheLookAndFeel( int value ) {
    try {
        UIManager.setLookAndFeel(
                looks[ value ].getClassName() );
        SwingUtilities.updateComponentTreeUI( this );
    }
    catch ( Exception exception ) {
        exception.printStackTrace();
    }
}
All the GUI components and event handling in this example have been covered before, so we concentrate on the mechanism for changing the look-and-feel in this example.
Class `UIManager` (package `javax.swing`) contains `public static` inner class `LookAndFeelInfo` that is used to maintain information about a look-and-feel. Line 14 declares an array of type `UIManagerLookAndFeelInfo` (notice the syntax used to access the inner class `LookAndFeelInfo`). Line 71 uses `static` method `getInstalledLookAndFeels` of class `UIManagerLookAndFeelInfo` objects that describe the installed look-and-feels.

**Performance Tip 13.1**

Each look-and-feel is represented by a Java class. `UIManager` method `getInstalledLookAndFeels` does not load each class. Rather, it provides access to the names of each look-and-feel, so a choice of look-and-feel can be made (presumably one time at program start-up). This reduces the overhead of loading additional classes that the program will not use.

Utility method `changeTheLookAndFeel` (lines 80–93) is called by the event handler (defined in `private` inner class `ItemHandler` at lines 105–121) for the `JRadioButton` at the bottom of the user interface. The event handler passes an integer representing the element in array `looks` that should be used to change the look-and-feel. Lines 84–85 use `static` method `setLookAndFeel` of class `UIManager` to change the look-and-feel. Method `getClassname` of class `UIManagerLookAndFeelInfo` determines the name of the look-and-feel class that corresponds to the `UIManagerLookAndFeelInfo`. If the look-and-feel class is not already loaded, it will be loaded as part of the call to `setLookAndFeel`. Line 86 uses `static` method `updateComponentTreeUI` of class `SwingUtilities` (package `javax.swing`) to change the look-and-feel of every component attached to its argument (this instance of class `LookAndFeelDemo`) to the new look-and-feel.

The preceding two statements appear in a special block of code called a **try block**. This code is part of the **exception-handling mechanism** discussed in detail in the next chapter. This code is required in case lines 84–85 attempt to change the look-and-feel to a look-and-feel that does not exist. Lines 90–92 complete the exception-handling mechanism with a **catch handler** that simply processes this problem (if it occurs) by printing an error message at the command line.

### 13.11 Using `JDesktopPane` and `JInternalFrame`

Many of today’s applications use a **multiple document interface (MDI)** (i.e., a main window (often called the parent window) containing other windows (often called child windows)) to manage several open documents that are being processed in parallel. For example, many e-mail programs allow you to have several e-mail windows open at the same time so you can compose and/or read multiple e-mail messages. Similarly, many word processors allow the user to open multiple documents in separate windows so the user can switch between the documents without having to close the current document to open another document. The program of Fig. 13.13 demonstrates Swing’s `JDesktopPane` and `JInternalFrame` classes, which provide support for creating multiple document interfaces. The child windows simply display an image of the cover of this book.

Lines 20–27 define a `JMenuBar`, a `JMenu` and a `JMenuItem`, add the `JMenuItem` to the `JMenu`, add the `JMenu` to the `JMenuBar` and set the `JMenuBar` for the application window. When the user selects the `JMenuItem newFrame`, the program creates and displays a new `JInternalFrame`. 
```
// Fig. 13.13: DesktopTest.java
// Demonstrating JDesktopPane.

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

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

public class DesktopTest extends JFrame {
    private JDesktopPane theDesktop;

    // set up GUI
    public DesktopTest()
        { super( "Using a JDesktopPane" );}

    // create menu bar, menu and menu item
    JMenuBar bar = new JMenuBar();
    JMenu addMenu = new JMenu( "Add" );
    JMenuItem newFrame = new JMenuItem( "Internal Frame" );

    addMenu.add( newFrame );
    bar.add( addMenu );

    setJMenuBar( bar );

    // set up desktop
    theDesktop = new JDesktopPane();
    getContentPane().add( theDesktop );

    // set up listener for newFrame menu item
    newFrame.addActionListener(
        new ActionListener() {
            // anonymous inner class to handle menu item event
            public void actionPerformed( ActionEvent event ) {
                // create internal frame
                JInternalFrame frame = new JInternalFrame(
                    "Internal Frame", true, true, true, true );

                // attach panel to internal frame content pane
                Container container = frame.getContentPane();
                MyJPanel panel = new MyJPanel();
                container.add( panel, BorderLayout.CENTER );

                // set size internal frame to size of its contents
                frame.pack();
            }
        });
}
```

Fig. 13.13 Creating a multiple document interface (part 1 of 3).
// attach internal frame to desktop and show it
theDesktop.add( frame );
frame.setVisible( true );
}
}  // end anonymous inner class
);  // end call to addActionListener
setSize( 600, 440 );
setVisible( true );
}  // end constructor

// execute application
public static void main( String args[] )
{
    DesktopTest application = new DesktopTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
}  // end class DesktopTest

// class to display an ImageIcon on a panel
class MyJPanel extends JPanel {
    private ImageIcon icon;

    // load image
    public MyJPanel()
    {
        icon = new ImageIcon( "jhtp4.png" );
    }

    // display icon on panel
    public void paintComponent( Graphics g )
    {
        // call superclass paintComponent method
        super.paintComponent( g );

        // display icon
        icon.paintIcon( this, g, 0, 0 );
    }

    // return image dimensions
    public Dimension getPreferredSize()
    {
        return new Dimension( icon.getIconWidth(),
                              icon.getIconHeight() );
    }
}  // end class MyJPanel

Fig. 13.13  Creating a multiple document interface (part 2 of 3).
Fig. 13.13 Creating a multiple document interface (part 3 of 3).
Line 30 creates `JDesktopPane` (package `javax.swing`) reference the `Desktop` and assigns it a new `JDesktopPane` object. The `JDesktopPane` object manages the `JInternalFrame` child windows displayed in the `JDesktopPane`. Line 31 adds the `JDesktopPane` to the application window’s content pane.

Lines 34–61 register an instance of an anonymous inner class that implements `ActionListener` to handle the event when the user selects the `newFrame` menu item. When the event occurs, method `actionPerformed` (lines 40–57) creates a `JInternalFrame` object with lines 43–44. The `JInternalFrame` constructor used here requires five arguments—a `String` for the title bar of the internal window, a `boolean` indicating whether the internal frame should be resizable by the user, a `boolean` indicating whether the internal frame should be closable by the user, a `boolean` indicating whether the internal frame should be maximizable by the user and a `boolean` indicating whether the internal frame should be minimizable by the user. For each of the boolean arguments, a `true` value indicates that the operation should be allowed.

As with `JFrame` and `JApplet`, a `JInternalFrame` has a content pane to which GUI components can be attached. Line 47 gets a reference to the `JInternalFrame`’s content pane. Line 48 creates an instance of our class `MyJPanel` (defined at lines 80–106) that is added to the `JInternalFrame`’s content pane at line 49.

Line 52 uses `JInternalFrame` method `pack` to set the size of the child window. Method `pack` uses the preferred sizes of the components on the content pane to determine the window’s size. Class `MyJPanel` defines method `getPreferredSize` to specify the panel’s preferred size. Line 55 adds the `JInternalFrame` to the `JDesktopPane`, and line 56 displays the `JInternalFrame`.

Classes `JInternalFrame` and `JDesktopPane` provide many methods for managing child windows. See the on-line API documentation for a complete list of these methods.

### 13.12 Layout Managers

In the preceding chapter, we introduced three layout managers—`FlowLayout`, `BorderLayout` and `GridLayout`. This section presents three additional layout managers (summarized in Figure 13.14). We discuss these layout managers in the sections that follow.

<table>
<thead>
<tr>
<th>Layout Manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoxLayout</td>
<td>A layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class <code>Box</code> defines a container with <code>BoxLayout</code> as its default layout manager and provides static methods to create a <code>Box</code> with a horizontal or vertical <code>BoxLayout</code>.</td>
</tr>
<tr>
<td>CardLayout</td>
<td>A layout manager that stacks components like a deck of cards. If a component in the deck is a container, it can use any layout manager. Only the component at the “top” of the deck is visible.</td>
</tr>
<tr>
<td>GridBagLayout</td>
<td>A layout manager similar to <code>GridLayout</code>. Unlike <code>GridLayout</code>, each component size can vary and components can be added in any order.</td>
</tr>
</tbody>
</table>

**Fig. 13.14** Additional layout managers.
13.13 BoxLayout Layout Manager

The **BoxLayout layout manager** arranges GUI components horizontally along the x-axis or vertically along the y-axis of a container. The program of Fig. 13.15 demonstrates **BoxLayout** and the container class **Box** that uses **BoxLayout** as its default layout manager.

In the constructor for class **BoxLayoutDemo**, lines 19–20 obtain a reference to the content pane and set its layout to a **BorderLayout** with 30 pixels of horizontal and 30 pixels of vertical gap space between components. The space is to help isolate each of the containers with **BoxLayout** in this example.

Lines 23–28 define an array of **Box** container references called **boxes** and initialize each element of the array with **Box** objects. Elements 0 and 2 of the array are initialized with static method **createHorizontalBox** of class **Box**, which returns a **Box** container with a horizontal **BoxLayout** (GUI components are arranged left-to-right). Elements 1 and 3 of the array are initialized with static method **createVerticalBox** of class **Box**, which returns a **Box** container with a vertical **BoxLayout** (GUI components are arranged top-to-bottom).

```java
// Fig. 13.15: BoxLayoutDemo.java
// Demonstrating BoxLayout.

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

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

public class BoxLayoutDemo extends JFrame {
    // set up GUI
    public BoxLayoutDemo()
    {
        super( "Demonstrating BoxLayout" );
        final int SIZE = 3;

        Container container = getContentPane();
        container.setLayout( new BorderLayout( 30, 30 ) );

        // create Box containers with BoxLayout
        Box boxes[] = new Box[ 4 ];

        boxes[ 0 ] = Box.createHorizontalBox();
        boxes[ 1 ] = Box.createVerticalBox();
        boxes[ 2 ] = Box.createHorizontalBox();
        boxes[ 3 ] = Box.createVerticalBox();

        // add buttons to boxes[ 0 ]
        for ( int count = 0; count < SIZE; count++ )
            boxes[ 0 ].add( new JButton( "boxes[0]: " + count ) );
    }
}
```

Fig. 13.15 Demonstrating the **BoxLayout** layout manager (part 1 of 3).
// create strut and add buttons to boxes[ 1 ]
for ( int count = 0; count < SIZE; count++ ) {
    boxes[ 1 ].add( Box.createVerticalStrut( 25 ) );
    boxes[ 1 ].add( new JButton( "boxes[1]: " + count ) );
}

// create horizontal glue and add buttons to boxes[ 2 ]
for ( int count = 0; count < SIZE; count++ ) {
    boxes[ 2 ].add( Box.createHorizontalGlue() );
    boxes[ 2 ].add( new JButton( "boxes[2]: " + count ) );
}

// create rigid area and add buttons to boxes[ 3 ]
for ( int count = 0; count < SIZE; count++ ) {
    boxes[ 3 ].add( Box.createRigidArea( new Dimension( 12, 8 ) ) );
    boxes[ 3 ].add( new JButton( "boxes[3]: " + count ) );
}

// create vertical glue and add buttons to panel
JPanel panel = new JPanel();
panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );
for ( int count = 0; count < SIZE; count++ ) {
    panel.add( Box.createGlue() );
    panel.add( new JButton( "panel: " + count ) );
}

// place panels on frame
container.add( boxes[ 0 ], BorderLayout.NORTH );
container.add( boxes[ 1 ], BorderLayout.EAST );
container.add( boxes[ 2 ], BorderLayout.SOUTH );
container.add( boxes[ 3 ], BorderLayout.WEST );
container.add( panel, BorderLayout.CENTER );
setSize( 350, 300 );
setVisible( true );

} // end constructor

// execute application
public static void main( String args[] )
{
    BoxLayoutDemo application = new BoxLayoutDemo();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
} // end class BoxLayoutDemo

Fig. 13.15 Demonstrating the BoxLayout layout manager (part 2 of 3).
The `for` structure at lines 31–32 adds three `JButton`s to `boxes[0]` (a horizontal `Box`). The `for` structure at lines 35–38 adds three `JButton`s to `boxes[1]` (a vertical `Box`). Before adding each button, line 36 adds a `vertical strut` to the container with `static` method `createVerticalStrut` of class `Box`. A vertical strut is an invisible GUI component that has a fixed pixel height and is used to guarantee a fixed amount of space between GUI components. The argument to method `createVerticalStrut` determines the height of the strut in pixels. Class `Box` also defines method `createHorizontalStrut` for horizontal `BoxLayouts`.

The `for` structure at lines 41–44 adds three `JButton`s to `boxes[2]` (a horizontal `Box`). Before adding each button, line 42 adds `horizontal glue` to the container with `static` method `createHorizontalGlue` of class `Box`. Horizontal glue is an invis-
ible GUI component that can be used between fixed-size GUI components to occupy additional space. Normally, extra space appears to the right of the last horizontal GUI component or below the last vertical GUI component in a BoxLayout. Glue allows the extra space to be placed between GUI components. Class Box also defines method createVerticalGlue for vertical BoxLayouts.

The for structure at lines 47–51 adds three JButtons to boxes[3] (a vertical Box). Before adding each button, lines 48–49 add a rigid area to the container with static method createRigidArea of class Box. A rigid area is an invisible GUI component that always has a fixed pixel width and height. The argument to method createRigidArea is a Dimension object that specifies the width and height of the rigid area.

Lines 54–56 create a JPanel object and set its layout in the conventional manner, using Container method setLayout. The BoxLayout constructor receives a reference to the container for which it controls the layout and a constant indicating whether the layout is horizontal (BoxLayout.X_AXIS) or vertical (BoxLayout.Y_AXIS).

The for structure at lines 58–61 adds three JButtons to panel. Before adding each button, line 59 adds a glue component to the container with static method createGlue of class Box. This component expands or contracts based on the size of the Box.

The Box containers and the JPanel are attached to the content pane’s BorderLayout at lines 64–68. Try executing the application. When the window appears, resize the window to see how the glue components, strut components and rigid area affect the layout in each container.

13.14 CardLayout Layout Manager

The CardLayout layout manager arranges components into a “deck” of cards where only the top card is visible. Any card in the deck can be placed at the top of the deck at any time by using methods of class CardLayout. Each card is usually a container, such as a panel, and each card can use any layout manager. Class CardLayout inherits from Object and implements the LayoutManager2 interface.

The program of Fig. 13.16 creates five panels. JPanel deck uses the CardLayout layout manager to control the card that is displayed. JPanels card1, card2 and card3 are the individual cards in deck. JPanel buttons contains four buttons (with labels First card, Next card, Previous card and Last card) that enable the user to manipulate the deck. When the user clicks the First card button, the first card in deck (i.e., card1) is displayed. When the user clicks the Last card button, the last card (i.e., card3) in deck is displayed. Each time the user clicks the Previous card button, the previous card in deck is displayed. Each time the user clicks the Next card button, the next card in deck is displayed. Clicking the Previous card button or the Next card button repeatedly allows the user to cycle through the deck of cards. Application class CardDeck implements ActionListener, so the action events generated by the JButtons on JPanel buttons are handled by the application in its actionPerformed method.

Class CardDeck declares a reference of type CardLayout called cardManager (line 13). This reference is used to invoke CardLayout methods that manipulate the cards in the deck.
// Fig. 13.16: CardDeck.java
// Demonstrating CardLayout.

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

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

public class CardDeck extends JFrame implements ActionListener {

    private CardLayout cardManager;
    private JPanel deck;
    private JButton controls[];
    private String names[] = { "First card", "Next card",
                                "Previous card", "Last card" };

    // set up GUI
    public CardDeck() {
        super( "CardLayout " );
        Container container = getContentPane();

        // create the JPanel with CardLayout
        deck = new JPanel();
        cardManager = new CardLayout();
        deck.setLayout( cardManager );

        // set up card1 and add it to JPanel deck
        JLabel label1 =
                new JLabel( "card one", SwingConstants.CENTER );
        JPanel card1 = new JPanel();
        card1.add( label1 );
        deck.add( card1, label1.getText() ); // add card to deck

        // set up card2 and add it to JPanel deck
        JLabel label2 =
                new JLabel( "card two", SwingConstants.CENTER );
        JPanel card2 = new JPanel();
        card2.setBackground( Color.yellow );
        card2.add( label2 );
        deck.add( card2, label2.getText() ); // add card to deck

        // set up card3 and add it to JPanel deck
        JLabel label3 = new JLabel( "card three" );
        JPanel card3 = new JPanel();
        card3.setLayout( new BorderLayout() );
        card3.add( new JButton( "North" ), BorderLayout.NORTH );
        card3.add( new JButton( "West" ), BorderLayout.WEST );
        card3.add( new JButton( "East" ), BorderLayout.EAST );
        card3.add( new JButton( "South" ), BorderLayout.SOUTH );
    }
}

Fig. 13.16 Demonstrating the CardLayout layout manager (part 1 of 3).
card3.add( label3, BorderLayout.CENTER );
deck.add( card3, label3.getText() ); // add card to deck

// create and layout buttons that will control deck
JPanel buttons = new JPanel();
bUTTONS.setLayout( new GridLayout( 2, 2 ) );
controls = new JButton[ names.length ];

for ( int count = 0; count < controls.length; count++ ) {
    controls[ count ] = new JButton( names[ count ] );
    controls[ count ].addActionListener( this );
    buttons.add( controls[ count ] );
}

// add JPanel deck and JPanel buttons to the applet
container.add( buttons, BorderLayout.WEST );
container.add( deck, BorderLayout.EAST );

setSize( 450, 200 );
setVisible( true );
}

// handle button events by switching cards
public void actionPerformed( ActionEvent event )
{
    // show first card
    if ( event.getSource() == controls[ 0 ] )
        cardManager.first( deck );

    // show next card
    else if ( event.getSource() == controls[ 1 ] )
        cardManager.next( deck );

    // show previous card
    else if ( event.getSource() == controls[ 2 ] )
        cardManager.previous( deck );

    // show last card
    else if ( event.getSource() == controls[ 3 ] )
        cardManager.last( deck );
}

// execute application
public static void main( String args[] )
{
    CardDeck cardDeckDemo = new CardDeck();
    cardDeckDemo.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

} // end class CardDeck

Fig. 13.16 Demonstrating the CardLayout layout manager (part 2 of 3).
The constructor method (lines 20–75) builds the GUI. Lines 27–29 create \textit{JPanel deck}, create \textit{CardLayout} object \textit{cardManager} and set the layout manager for \textit{deck} to \textit{cardManager}. Next, the constructor creates the \textit{JPanel}s \textit{card1}, \textit{card2}, and \textit{card3} and their GUI components. As we set up each card, we add the card to \textit{deck}, using \textit{Container} method \textit{add} with two arguments—a \texttt{Component} and a \texttt{String}. The \texttt{Component} is the \textit{JPanel} object that represents the card. The \texttt{String} argument identifies the card. For example, line 36 adds \texttt{JPanel card1} to \textit{deck} and uses \texttt{JLabel label1}’s label as the \texttt{String} identifier for the card. \texttt{JPanel}s \textit{card2} and \textit{card3} are added to \textit{deck} at lines 44–55. Next, the constructor creates the \textit{JPanel buttons} and its \textit{JButton} objects (lines 58–66). Line 64 registers the \texttt{ActionListener} for each \texttt{JButton}. Finally, \texttt{buttons} and \textit{deck} are added to the content pane’s \texttt{WEST} and \texttt{EAST} regions, respectively.

Method \texttt{actionPerformed} (lines 78–95) determines which \texttt{JButton} generated the event by using \texttt{EventObject} method \texttt{getSource}. \texttt{CardLayout} methods \texttt{first}, \texttt{previous}, \texttt{next} and \texttt{last} are used to display the particular card corresponding to which \texttt{JButton} the user pressed. Method \texttt{first} displays the first card added to the deck. Method \texttt{previous} displays the previous card in the deck. Method \texttt{next} displays the next card in the deck. Method \texttt{last} displays the last card in the deck. Note that \textit{deck} is passed to each of these methods.
13.15 GridBagLayout Layout Manager

The most complex and most powerful of the predefined layout managers is GridBagLayout. This layout is similar to GridLayout because GridBagLayout also arranges components in a grid. However, GridBagLayout is more flexible. The components can vary in size (i.e., they can occupy multiple rows and columns) and can be added in any order.

The first step in using GridBagLayout is determining the appearance of the GUI. This step does not involve any programming; all that is needed is a piece of paper. First, draw the GUI. Next draw a grid over the GUI dividing the components into rows and columns. The initial row and column numbers should be 0 so the GridBagLayout layout manager can properly place the components in the grid. The row and column numbers will be used to place each component in an exact position in the grid. Figure 13.17 demonstrates drawing the lines for the rows and columns over a GUI.

To use GridBagLayout, a GridBagConstraints object must be constructed. This object specifies how a component is placed in a GridBagLayout. Several GridBagConstraints instance variables are summarized in Fig. 13.18.

<table>
<thead>
<tr>
<th>GridBagConstraints Instance Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridx</td>
<td>The column in which the component will be placed.</td>
</tr>
<tr>
<td>gridy</td>
<td>The row in which the component will be placed.</td>
</tr>
<tr>
<td>gridwidth</td>
<td>The number of columns the component occupies.</td>
</tr>
<tr>
<td>gridheight</td>
<td>The number of rows the component occupies.</td>
</tr>
<tr>
<td>weightx</td>
<td>The portion of extra space to allocate horizontally. The components can become wider when extra space is available.</td>
</tr>
<tr>
<td>weighty</td>
<td>The portion of extra space to allocate vertically. The components can become taller when extra space is available.</td>
</tr>
</tbody>
</table>

![Fig. 13.17](image-url) Designing a GUI that will use GridBagLayout.

![Fig. 13.18](image-url) GridBagConstraints instance variables.
Variables \textit{gridx} and \textit{gridy} specify the row and column where the upper-left corner of the component is placed in the grid. Variable \textit{gridx} corresponds to the column and the variable \textit{gridy} corresponds to the row. In Fig. 13.17, the \texttt{JComboBox} (displaying “Iron”) has a \textit{gridx} value of 1 and a \textit{gridy} value of 2.

Variable \textit{gridwidth} specifies the number of columns a component occupies. In Fig. 13.17, the \texttt{JComboBox} button occupies two columns. Variable \textit{gridheight} specifies the number of rows a component occupies. In Fig. 13.17, the \texttt{JTextArea} on the left side of the window occupies three rows.

Variable \textit{weightx} specifies how to distribute extra horizontal space to components in a \texttt{GridBagLayout} when the container is resized. A zero value indicates that the component does not grow horizontally on its own. However, if the component spans a column containing a component with nonzero \textit{weightx} value, the component with zero \textit{weightx} value will grow horizontally in the same proportion as the other component(s) in the same column. This is because each component must be maintained in the same row and column in which it was originally placed.

Variable \textit{weighty} specifies how to distribute extra vertical space to components in a \texttt{GridBagLayout} when the container is resized. A zero value indicates that the component does not grow vertically on its own. However, if the component spans a row containing a component with nonzero \textit{weighty} value, the component with zero \textit{weighty} value grows vertically in the same proportion as the other component(s) in the same row.

In Fig. 13.17, the effects of \textit{weighty} and \textit{weightx} cannot easily be seen until the container is resized and additional space becomes available. Components with larger weight values occupy more of the additional space than components with smaller weight values. The exercises explore the effects of varying \textit{weightx} and \textit{weighty}.

Components should be given nonzero positive weight values—otherwise the components will “huddle” together in the middle of the container. Figure 13.19 shows the GUI of Fig. 13.17—where all weights have been set to zero.

\textbf{Common Programming Error 13.6}

Using a negative value for either \textit{weightx} or \textit{weighty} is a logic error.

\texttt{GridBagConstraints} instance variable \textit{fill} specifies how much of the component’s area (the number of rows and columns the component occupies in the grid) is occupied. The variable \textit{fill} is assigned one of the following \texttt{GridBagConstraints} constants: \texttt{NONE}, \texttt{VERTICAL}, \texttt{HORIZONTAL} or \texttt{BOTH}. The default value is \texttt{NONE}, which indicates that the component will not grow in either direction. \texttt{VERTICAL} indicates that the component will grow vertically. \texttt{HORIZONTAL} indicates that the component will grow horizontally. \texttt{BOTH} indicates that the component will grow in both directions.

\texttt{GridBagConstraints} instance variable \textit{anchor} specifies the location of the component in the area when the component does not fill the entire area. The variable \textit{anchor} is assigned one of the following \texttt{GridBagConstraints} constants: \texttt{NORTH}, \texttt{NORTHEAST}, \texttt{EAST}, \texttt{SOUTHEAST}, \texttt{SOUTH}, \texttt{SOUTHWEST}, \texttt{WEST}, \texttt{NORTHWEST} or \texttt{CENTER}. The default value is \texttt{CENTER}.

The program of Fig. 13.20 uses the \texttt{GridBagLayout} layout manager to arrange the components in the GUI of Fig. 13.17. The program does nothing other than demonstrate how to use \texttt{GridBagLayout}. 
Fig. 13.19 *GridBagLayout* with the weights set to zero.

Fig. 13.20 Demonstrating the *GridBagLayout* layout manager (part 1 of 3).

```java
// Fig. 13.20: GridBagDemo.java
// Demonstrating GridBagLayout.

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

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

public class GridBagDemo extends JFrame {

    // set up GUI
    public GridBagDemo() {
        super( "GridBagLayout" );
        container = getContentPane();
        layout = new GridBagLayout();
        container.setLayout( layout );
        // instantiate gridbag constraints
        constraints = new GridBagConstraints();

        // create GUI components
        JTextArea textArea1 = new JTextArea( "TextArea1", 5, 10 );
        JTextArea textArea2 = new JTextArea( "TextArea2", 2, 2 );
        String names[] = { "Iron", "Steel", "Brass" };
        JComboBox comboBox = new JComboBox( names );

        JTextField textField = new JTextField( "TextField" );
        JButton button1 = new JButton( "Button 1" );
        JButton button2 = new JButton( "Button 2" );
        JButton button3 = new JButton( "Button 3" );
```
Chapter 13  Graphical User Interface Components: Part 2

Fig. 13.20  Demonstrating the GridBagLayout layout manager (part 2 of 3).

```java
// textArea1
// weightx and weighty are both 0: the default
// anchor for all components is CENTER: the default
constraints.fill = GridBagConstraints.BOTH;
addComponent( textArea1, 0, 0, 1, 3 );

// button1
// weightx and weighty are both 0: the default
constraints.fill = GridBagConstraints.HORIZONTAL;
addComponent( button1, 0, 1, 2, 1 );

// comboBox
// weightx and weighty are both 0: the default
// fill is HORIZONTAL
addComponent( comboBox, 2, 1, 2, 1 );

// button2
// fill is BOTH
constraints.weightx = 1000;  // can grow wider
constraints.weighty = 1;     // can grow taller
constraints.fill = GridBagConstraints.BOTH;
addComponent( button2, 1, 1, 1, 1 );

// button3
// fill is BOTH
constraints.weightx = 0;
constraints.weighty = 0;
addComponent( button3, 1, 2, 1, 1 );

// textField
// weightx and weighty are both 0, fill is BOTH
addComponent( textField, 3, 0, 2, 1 );

// textArea2
// weightx and weighty are both 0, fill is BOTH
addComponent( textArea2, 3, 2, 1, 1 );

setSize( 300, 150 );
setVisible( true );

// method to set constraints on
private void addComponent( Component component,
int row, int column, int width, int height )
{
    // set gridx and gridy
    constraints.gridx = column;
    constraints.gridy = row;

    // set gridwidth and gridheight
    constraints.gridwidth = width;
    constraints.gridheight = height;
}
```

Fig. 13.20  Demonstrating the GridBagLayout layout manager (part 2 of 3).
The GUI consists of three JButton, two JTextArea, a JComboBox and a JTextField. The layout manager for the content pane is GridBagLayout. Lines 22–23 instantiate the GridBagLayout object and set the layout manager for the content pane to layout. The GridBagConstraints object used to determine the location and size

Fig. 13.20 Demonstrating the GridBagLayout layout manager (part 3 of 3).
of each component in the grid is instantiated with line 26. Lines 23 through 30 instantiate each of the GUI components that will be added to the content pane.

**JTextArea** textArea1 is the first component added to the `GridBagLayout` (line 44). The values for `weightx` and `weighty` values are not specified in `gbConstraints`, so each has the value zero by default. Thus, the **JTextArea** will not resize itself even if space is available. However, the **JTextArea** spans multiple rows, so the vertical size is subject to the `weighty` values of **JButtons** button2 and button3. When either button2 or button3 is resized vertically based on its `weighty` value, the **JTextArea** is also resized.

Line 43 sets variable `fill` in `constraints` to `GridBagConstraints.BOTH`, causing the **JTextArea** to always fill its entire allocated area in the grid. An `anchor` value is not specified in `constraints`, so the default CENTER is used. We do not use variable `anchor` in this program, so all components will use the default. Line 36 calls our utility method `addComponent` method (defined at lines 81–95). The **JTextArea** object, the row, the column, the number of columns to span and the number of rows to span are passed as arguments.

Method `addComponent`’s parameters are a **Component** reference component and integers `row`, `column`, `width` and `height`. Lines 85–86 set the `GridBagConstraints` variables `gridx` and `gridy`. The `gridx` variable is assigned the column in which the **Component** will be placed, and the `gridy` value is assigned the row in which the **Component** will be placed. Lines 89–90 set the `GridBagConstraints` variables `gridwidth` and `gridheight`. The `gridwidth` variable specifies the number of columns the **Component** will span in the grid and the `gridheight` variable specifies the number of rows the **Component** will span in the grid. Line 93 sets the `GridBagConstraints` for a component in the `GridBagLayout`. Method `setConstraints` of class `GridBagLayout` takes a **Component** argument and a `GridBagConstraints` argument. Method `add` (line 94) is used to add the component to the content pane.

**JButton** object button1 is the next component added (lines 48–49). The values of `weightx` and `weighty` are still zero. The `fill` variable is set to `HORIZONTAL`—the component will always fill its area in the horizontal direction. The vertical direction is not filled. The `weighty` value is zero, so the button will become taller only if another component in the same row has a nonzero `weighty` value. **JButton** b1 is located at row 0, column 1. One row and two columns are occupied.

**JComboBox** comboBox is the next component added (line 54). The `weightx` and `weighty` values are zero, and the `fill` variable is set to `HORIZONTAL`—the component will always fill its area in the horizontal direction. The vertical direction is not filled. The `weighty` value is zero, so the button will become taller only if another component in the same row has a nonzero `weighty` value. **JButton** b1 is located at row 0, column 1. One row and two columns are occupied.

**JComboBox** comboBox is the next component added (line 54). The `weightx` and `weighty` values are zero, and the `fill` variable is set to `HORIZONTAL`. The **JComboBox** button will grow only in the horizontal direction. Note that the `weightx`, `weighty` and `fill` variables remain set in `gbConstraints` until they are changed. The **JComboBox** button is placed at row 2, column 1. One row and two columns are occupied.

**JButton** object button2 is the next component added (lines 57–60). **JButton** button2 is given a `weightx` value of 1000 and a `weighty` value of 1. The area occupied by the button is capable of growing in the vertical and horizontal directions. The `fill` variable is set to `BOTH`, which specifies that the button will always fill the entire area. When the window is resized, b2 will grow. The button is placed at row 1, column 1. One row and one column are occupied.

**JButton** button3 is added next (lines 64–66). Both the `weightx` value and `weighty` value are set to zero, and the value of `fill` is `BOTH`. **JButton** button3 will
grow if the window is resized; it is affected by the weight values of `button2`. Note that the `weightx` value for `button2` is much larger than `button3`. When resizing occurs, `button2` will occupy a larger percentage of the new space. The button is placed at row 1, column 2. One row and one column are occupied.

Both the `JTextField` (line 70) and `JTextArea textArea2` (line 74) have a `weightx` value 0 and a `weighty` value 0. The value of `fill` is `BOTH`. The `JTextField` is placed at row 3, column 0, and the `JTextArea` is placed at row 3, column 2. The `JTextField` occupies one row and two columns. The `JTextArea` occupies one row and one column.

When you execute this application, try resizing the window to see how the constraints for each GUI component affect its position and size in the window.

### 13.16 GridBagConstraints Constants RELATIVE and REMAINDER

A variation of `GridBagLayout` does not use `gridx` and `gridy`. Rather, `GridBagConstraints` constants `RELATIVE` and `REMAINDER` are used in their place. `RELATIVE` specifies that the next-to-last component in a particular row should be placed to the right of the previous component in that row. `REMAINDER` specifies that a component is the last component in a row. Any component that is not the second-to-last or last component on a row must specify values for `GridBagConstraints` variables `gridwidth` and `gridheight`. Class `GridBagDemo2` in Fig. 13.21 arranges components in `GridBagLayout`, using these constants.

```java
// Fig. 13.21: GridBagDemo2.java
// Demonstrating GridBagConstraints constants RELATIVE and REMAINDER.

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

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

// Fig. 13.21: GridBagDemo2.java
// Demonstrating GridBagConstraints constants RELATIVE and REMAINDER.

public class GridBagDemo2 extends JFrame {
    private GridBagLayout layout;
    private GridBagConstraints constraints;
    private Container container;

    // set up GUI
    public GridBagDemo2() {
        super( "GridBagLayout" );
        container = getContentPane();
        layout = new GridBagLayout();
        container.setLayout( layout );
    }
}
```

**Fig. 13.21** Demonstrating the `GridBagConstraints` constants `RELATIVE` and `REMAINDER` (part 1 of 3).
// instantiate gridbag constraints
constraints = new GridBagConstraints();

// create GUI components
String metals[] = { "Copper", "Aluminum", "Silver" };
JComboBox comboBox = new JComboBox( metals );

JTextField textField = new JTextField( "TextField" );

String fonts[] = { "Serif", "Monospaced" };
JList list = new JList( fonts );

String names[] =
{ "zero", "one", "two", "three", "four" };
JButton buttons[] = new JButton[ names.length ];

for ( int count = 0; count < buttons.length; count++ )
    buttons[ count ] = new JButton( names[ count ] );

// define GUI component constraints
// textField
constraints.weightx = 1;
constraints.weighty = 1;
constraints.fill = GridBagConstraints.BOTH;
constraints.gridwidth = GridBagConstraints.REMAINDER;
addComponent( textField );

// buttons[0] -- weightx and weighty are 1: fill is BOTH
constraints.gridwidth = 1;
addComponent( buttons[ 0 ] );

// buttons[1] -- weightx and weighty are 1: fill is BOTH
constraints.gridwidth = GridBagConstraints.RELATIVE;
addComponent( buttons[ 1 ] );

// buttons[2] -- weightx and weighty are 1: fill is BOTH
constraints.gridwidth = GridBagConstraints.REMAINDER;
addComponent( buttons[ 2 ] );

// comboBox -- weightx is 1: fill is BOTH
constraints.weighty = 0;
constraints.gridwidth = GridBagConstraints.REMAINDER;
addComponent( comboBox );

// buttons[3] -- weightx is 1: fill is BOTH
constraints.weighty = 1;
constraints.gridwidth = GridBagConstraints.REMAINDER;
addComponent( buttons[ 3 ] );

// buttons[4] -- weightx and weighty are 1: fill is BOTH
constraints.gridwidth = GridBagConstraints.RELATIVE;
addComponent( buttons[ 4 ] );
Lines 22–23 construct a `GridLayout` and set the content pane’s layout manager to `GridLayout`. The components that are placed in `GridLayout` are each constructed (lines 29–42). The components are five `JButton`, one `JTextField`, one `JList` and one `JComboBox`. The `JTextField` is added first (lines 46–50). The `weightx` and `weighty` values are set to 1. The `fill` variable is set to `BOTH`. Line 49 specifies that the `JTextField` is the last component on the line. The `JTextField` is added to the content pane with a call to our utility method `addComponent` (defined at lines 88–92). Method `addComponent` takes a `Component` argument and uses `GridLayout` method `setConstraints` to set the constraints for the `Component`. Method `add` attaches the component to the content pane.
**Chapter 13** Graphical User Interface Components: Part 2

JButton buttons[0] (lines 53–54) has weightx and weighty values of 1. The fill variable is BOTH. Because buttons[0] is not one of the last two components on the row, it is given a gridwidth of 1 so it will occupy one column. The JButton is added to the content pane with a call to utility method addComponent.

JButton buttons[1] (lines 57–58) has weightx and weighty values of 1. The fill variable is BOTH. Line 57 specifies that the JButton is to be placed relative to the previous component. The Button is added to the JFrame with a call to addComponent.

JButton buttons[2] (lines 61–62) has weightx and weighty values of 1. The fill variable is BOTH. This JButton is the last component on the line, so REMAINDER is used. The JButton is added to the content pane with a call to utility method addComponent.

The JComboBox button (lines 65–67) has a weightx 1 and a weighty 0. The JComboBox will not grow in the vertical direction. The JComboBox is the only component on the line, so REMAINDER is used. The JComboBox is added to the content pane with a call to utility method addComponent.

JButton buttons[3] (lines 70–72) has weightx and weighty values of 1. The fill variable is BOTH. This JButton is the only component on the line, so REMAINDER is used. The JButton is added to the content pane with a call to utility method addComponent.

JButton buttons[4] (lines 75–76) has weightx and weighty values of 1. The fill variable is BOTH. This JButton is the next-to-last component on the line, so RELATIVE is used. The JButton is added to the content pane with a call to utility method addComponent.

The JList component (lines 79–80) has weightx and weighty values of 1. The fill variable is BOTH. The JList is added to the content pane with a call to utility method addComponent.

### 13.17 (Optional Case Study) Thinking About Objects: Model-View-Controller

Design patterns describe proven strategies for building reliable object-oriented software systems. Our case study adheres to the Model-View-Controller (MVC) architecture, which uses several design patterns. MVC divides system responsibilities into three parts:

1. the model, which contains all program data and logic;
2. the view, which provides a visual presentation of the model and
3. the controller, which defines the system behavior by sending user input to the model.

Using the controller, the user changes the data in the model. The model then informs the view of the change in data. The view changes its visual presentation to reflect the changes in the model.

For example, in our simulation, the user adds a Person to the model by pressing either the First Floor or Second Floor JButton in the controller (see Fig. 2.22–

---

1. For those readers who seek further study in design patterns and MVC architecture, we encourage you to read our “Discovering Design Patterns” material in Sections 1.16, 9.24, 13.18, 15.13, 17.11 and 21.12
The model then notifies the view of the **Person**’s creation. The view, in response to this notification, displays a **Person** on a **Floor**. The model is unaware of how the view displays the **Person**, and the view is unaware of how or why the model created the **Person**.

The MVC architecture helps construct reliable and easily modifiable systems. If we desire text-based output rather than graphical output for the elevator simulation, we may create an alternate view to produce text-based output, without altering the model or the controller. We could also provide a three-dimensional view that uses a first-person perspective to allow the user to “take part” in the simulation; such views are commonly employed in virtual-reality-based systems.

**Elevator-Simulation MVC**

We now apply the MVC architecture to our elevator simulation. Every UML diagram we have provided to this point (with the exception of the use-case diagram) relates to the model of our elevator system. We provide a “higher-level” class diagram of the simulation in Fig. 13.19. Class **ElevatorSimulation**—a **JFrame** subclass—aggregates one instance each of classes **ElevatorModel**, **ElevatorView** and **ElevatorController** to create the **ElevatorSimulation** application. As mentioned in “Thinking About Objects” Section 10.22, the rectangle with the upper-right corner “folded over” represents a note in the UML. In this case, each note points to a specific class (through a dotted line) to describe that class’ role in the system. Classes **ElevatorModel**, **ElevatorView** and **ElevatorController** encapsulate all objects comprising the model, view and controller portions of our simulation, respectively.

---

**Fig. 13.19** Class diagram of the elevator simulation.
We showed in Fig. 9.38 that class \texttt{ElevatorModel} is an aggregation of several classes. To save space, we do not repeat this aggregation in Fig. 13.19. Class \texttt{ElevatorView} is also an aggregation of several classes—we expand the class diagram of \texttt{ElevatorView} in “Thinking About Objects” Section 22.9 to show these additional classes. Class \texttt{ElevatorController}, as described in Section 12.16, represents the simulation controller. Note that class \texttt{ElevatorView} implements interface \texttt{ElevatorModelListener}, which implements all interfaces used in our simulation, so the \texttt{ElevatorView} can receive all events from the model.

\textbf{Software Engineering Observation 13.7}

When appropriate, partition the system class diagram into several smaller class diagrams, so each diagram represents a unique subsystem.

Class \texttt{ElevatorSimulation} contains no attributes other than its references to an \texttt{ElevatorModel} object, an \texttt{ElevatorView} object and an \texttt{ElevatorController} object. The only behavior for class \texttt{ElevatorSimulation} is to start the program—therefore, in Java, class \texttt{ElevatorSimulation} contains a \texttt{static main} method that calls the constructor, which instantiates the \texttt{ElevatorModel}, \texttt{ElevatorView} and \texttt{ElevatorController} objects. We implement class \texttt{ElevatorSimulation} in Java later in this section.

\textbf{Component Diagrams}

Figure 13.19 helps us construct another diagram of the UML that we use to design our system—the component diagram. The component diagram models the “pieces”—called components—that the system needs to perform its tasks. These pieces include binary executables, compiled .\texttt{class} files, \texttt{.java} files, images, packages, resources, etc. We present the component diagram for our simulation in Fig. 13.20.

In Fig. 13.20, each box that contains the two small white boxes overlapping its left side is a component. A component in the UML is drawn similar to a plug (the two overlapping boxes represent the plug’s prongs)—a component may be “plugged-in” to other systems without having to change the component. Our system contains five components: \texttt{ElevatorSimulation.class}, \texttt{ElevatorSimulation.java}, \texttt{ElevatorModel.java}, \texttt{ElevatorView.java} and \texttt{ElevatorController.java}.

In Fig. 13.20, the graphics that resemble folders (boxes with tabs in their upper-left corners) represent packages in the UML. We can group classes, objects, components, use cases, etc., in a package. In this diagram (and in the remainder of our case study), the UML packages refer to Java packages (introduced in Section 8.5). In our discussion, we use lower-case bold-face Courier type for package names. The packages in our system are \texttt{model}, \texttt{view} and \texttt{controller}. Component \texttt{ElevatorSimulation.java} contains one instance each of all components in these packages. Currently, each package contains only one component—a \texttt{.java} file. The \texttt{model} package contains \texttt{ElevatorModel.java}, the \texttt{view} package contains \texttt{ElevatorView.java} and the \texttt{controller} package contains \texttt{ElevatorController.java}. We add components to each package in the appendices, when we implement each class from our class diagrams into a component (\texttt{.java} file).

The dotted arrows in Fig. 13.20 indicate a dependency between two components—the direction of the arrow indicates the “depends on” relationship. A dependency describes the...
relationship between components in which changes in one component affect another component. For example, component `ElevatorSimulation.class` depends on component `ElevatorSimulation.java`, because a change in `ElevatorSimulation.java` affects `ElevatorSimulation.class` when `ElevatorSimulation.java` is compiled. Section 12.16 mentioned that the `ElevatorController` object contains a reference to the `ElevatorModel` object (to place Persons on Floors). Therefore, `ElevatorController.java` depends on `ElevatorModel.java`.

Software Engineering Observation 13.8

A component diagram's dependencies help designers group components for reuse in future systems. For example, in our simulation, designers can reuse `ElevatorModel.java` in other systems without having to reuse `ElevatorView.java` (and vice versa), because these components do not depend on each other. However, if a designer wanted to reuse `ElevatorController.java`, the designer would have to reuse `ElevatorModel.java`.

![Component diagram for elevator simulation.](image)
According to Fig. 13.20, 

```
ElevatorModel.java and ElevatorView.java do not depend on each other—they communicate through interface ElevatorModelListener, which implements all interfaces in the simulation. ElevatorView.java realizes interface ElevatorModelListener, and ElevatorModel.java depends on interface ElevatorModelListener.
```

Figure 13.20 contains several stereotypes—words placed in guillemets (« ») indicating an element’s role. We mentioned the «interface» stereotype in “Thinking About Objects” Section 11.10. The «compilation» stereotype describes the dependency between ElevatorSimulation.class and ElevatorSimulation.java—ElevatorSimulation.java compiles to ElevatorSimulation.class. The «executable» stereotype specifies that a component is an application, and the «file» stereotype specifies that a component is a file containing source code for the executable.

**Implementation: ElevatorSimulation.java**

We use the component diagram of Fig. 13.20, the class diagram of Fig. 13.19 and the use-case diagram of Fig. 12.28 to implement *ElevatorSimulation.java* (Fig. 13.21). Lines 12–14 import packages *model*, *view* and *controller* as specified in Fig. 13.20.

```java
1 // ElevatorSimulation.java
2 // Application with Elevator Model, View, and Controller (MVC)
3 package com.deitel.jhtp4.elevator;
4
5 // Java core packages
6 import java.awt.*;
7
8 // Java extension packages
9 import javax.swing.*;
10
11 // Deitel packages
12 import com.deitel.jhtp4.elevator.model.*;
13 import com.deitel.jhtp4.elevator.view.*;
14 import com.deitel.jhtp4.elevator.controller.*;
15
16 public class ElevatorSimulation extends JFrame {
17    // model, view and controller
18    private ElevatorModel model;
19    private ElevatorView view;
20    private ElevatorController controller;
21
22    // constructor instantiates model, view, and controller
23    public ElevatorSimulation() {
24       super( "Deitel Elevator Simulation" );
25       // instantiate model, view and controller
26       model = new ElevatorModel();
27       view = new ElevatorView();
28       controller = new ElevatorController( model );
29    }
30}
```

*Fig. 13.21* Class *ElevatorSimulation* is the application for the elevator simulation (part 1 of 2).
The class diagram of Fig. 13.19 specifies that class `ElevatorSimulation` is a subclass of `javax.swing.JFrame`—line 16 declares class `ElevatorSimulation` as a public class extending class `JFrame`. Lines 19–21 implement class `ElevatorSimulation`’s aggregation of class `ElevatorModel`, the `ElevatorView` and the `ElevatorController` (shown in Fig. 13.19) by declaring one object from each class. Lines 29–31 of the `ElevatorSimulation` constructor initialize these objects.

Figure 13.19 and Fig. 13.20 specify that the `ElevatorView` is an `ElevatorModelListener` for the `ElevatorModel`. Line 34 registers the `ElevatorView` as a listener for `ElevatorModelEvent` so the `ElevatorView` can receive events from the `ElevatorModel` and properly represent the state of the model. Lines 37–38 add the `ElevatorView` and the `ElevatorController` to the `ElevatorSimulation`. According to the stereotypes in Fig. 13.20, `ElevatorSimulation.java` compiles to `ElevatorSimulation.class`, which is executable—lines 43–50 provide method `main` that runs the application.

We have completed the design of the components of our system. “Thinking About Objects” Section 15.12 concludes the design of the model by solving the interaction problems encountered in Fig. 10.26. Finally, Section 22.9 completes the design of the view and describes in greater detail how the `ElevatorView` receives events from the `ElevatorModel`. These last two sections will prepare you for the walkthrough of our elevator-simulation implementation in Appendices G, H and I.

### 13.18 (Optional) Discovering Design Patterns: Design Patterns Used in Packages `java.awt` and `javax.swing`

We continue our discussion from Section 9.24 on design patterns. This section introduces those design patterns associated with Java GUI components. After reading this section, you
should understand better how these components take advantage of design patterns and how developers integrate design patterns with Java GUI applications.

13.18.1 Creational Design Patterns

Now, we continue our treatment of creational design patterns, which provide ways to instantiate objects in a system.

**Factory Method**

Suppose we are designing a system that opens an image from a specified file. Several different image formats exist, such as GIF and JPEG. We can use method `createImage` of class `java.awt.Component` to create an `Image` object. For example, to create a JPEG and GIF image in an object of a `Component` subclass—such as a `JPanel` object, we pass the name of the image file to method `createImage`, which returns an `Image` object that stores the image data. We can create two `Image` objects, each which contains data for two images having entirely different structures. For example, a JPEG image can hold up to 16.7 million colors, whereas a GIF image can hold up to only 256. Also, a GIF image can contain transparent pixels that are not rendered on screen, whereas a JPEG image cannot contain transparent pixels.

Class `Image` is an abstract class that represents an image we can display on screen. Using the parameter passed by the programmer, method `createImage` determines the specific `Image` subclass from which to instantiate the `Image` object. We can design systems to allow the user to specify which image to create, and method `createImage` will determine the subclass from which to instantiate the `Image`. If the parameter passed to method `createImage` references a JPEG file, method `createImage` instantiates and returns an object of an `Image` subclass suitable for JPEG images. If the parameter references a GIF file, `createImage` instantiates and returns an object of an `Image` subclass suitable for GIF images.

Method `createImage` is an example of the Factory Method design pattern. The sole purpose of this factory method is to create objects by allowing the system to determine which class to instantiate at run time. We can design a system that allows a user to specify what type of image to create at run time. Class `Component` might not be able to determine which `Image` subclass to instantiate until the user specifies the image to load. For more information on method `createImage`, visit

www.java.sun.com/j2se/1.3/docs/api/java/awt/Component.html#createImage(java.awt.image.ImageProducer)

13.18.2 Structural Design Patterns

We now discuss three more structural design patterns. The Adapter design pattern helps objects with incompatible interfaces collaborate with one another. The Bridge design pattern helps designers enhance platform independence in their systems. The Composite design pattern provides a way for designers to organize and manipulate objects.

**Adapter**

The Adapter design pattern 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.

Java provides several classes that use the Adapter design pattern. Objects of these classes act as adapters between objects that generate certain events and those objects that handle these events. For example, a `MouseAdapter`, which we explained in Section 12.12, adapts an object that generates `MouseEvent` to an object that handles `MouseEvent`s.

**Bridge**

Suppose we are designing class `Button` 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 different 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 for 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` calls factory method `createButton` of class `Toolkit` to create the platform-specific `ButtonPeer` object. The `Button` object stores a reference to its `ButtonPeer`—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` to fulfill the request. If a designer creates a `Button` subclass called `ImageButton`, the designer does not need to create a corresponding `Win32ImageButton` or `MacImageButton` with platform-specific image-drawing capabilities. An `ImageButton` “is a” `Button`. Therefore, when an `ImageButton` needs to display its image, the `ImageButton` uses its `ButtonPeer`’s `Graphics` object to render the image on each platform. This design pattern enables designers to create new cross-platform GUI components using a “bridge” to hide platform-specific details.

**Portability Tip 13.2**

Designers often use the Bridge design pattern to enhance the platform independence of their systems. This design pattern enables designers to create new cross-platform components using a “bridge” to hide platform-specific details.
Composite

Designers often organize components into hierarchical structures (e.g., a hierarchy of directories and files on a hard drive)—each node in the structure represents a component (e.g., a file or directory). Each node can contain references to other nodes. A node is called a branch if it contains a reference to one or more nodes (e.g., a directory containing files). A node is called a leaf if it does not contain a reference to another node (e.g., a file). Occasionally, a structure contains objects from several different classes (e.g., a directory can contain files and directories). When an object—called a client—wants to traverse the structure, the client must determine the particular class for each node. Making this determination can be time consuming, and the structure can become hard to maintain.

In the Composite design pattern, each component in a hierarchical structure implements the same interface or extends a common superclass. This polymorphism (introduced in Section 9.10) ensures that clients can traverse all elements—branch or leaf—uniformly in the structure. Using this pattern, a client traversing the structure does not have to determine each component type, because all components implement the same interface or extend the same superclass.

Java GUI components use the Composite design pattern. Consider the Swing component class JPanel, which extends class JComponent. Class JComponent extends class java.awt.Container, which extends class java.awt.Component (Fig. 13.22). Class Container provides method add, which appends a Component object (or Component subclass object) to that Container object. Therefore, a JPanel object may be added to any object of a Component subclass, and any object from a Component subclass may be added to that JPanel object. A JPanel object can contain any GUI component while remaining unaware of that component’s specific type.

A client, such as a JPanel object, can traverse all components uniformly in the hierarchy. For example, if the JPanel object calls method repaint of superclass Container, method repaint displays the JPanel object and all components added to the JPanel object. Method repaint does not have to determine each component’s type, because all components inherit from superclass Container, which contains method repaint.

![Inheritance hierarchy for class JPanel.](image-url)
13.18.3 Behavioral Design Patterns

In this section, we continue our discussion on behavioral design patterns. We discuss the Chain-of-Responsibility, Command, Observer, Strategy and Template Method design patterns.

Chain-of-Responsibility

In object-oriented systems, objects interact by sending messages to one another. 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.

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.

Several Java AWT GUI components use the Chain-of-Responsibility design pattern to handle certain events. For example, class `java.awt.Button` overrides method `processEvent` of class `java.awt.Component` to process `AWTEvent` objects. Method `processEvent` attempts to handle the `AWTEvent` upon receiving this event as an argument. If method `processEvent` determines that the `AWTEvent` is an `ActionEvent` (i.e., the `Button` has been pressed), the method handles the event by invoking method `processActionEvent`, which informs any `ActionListener` registered with the `Button` that the `Button` has been pressed. If method `processEvent` determines that the `AWTEvent` is not an `ActionEvent`, the method is unable to handle the event and passes the `AWTEvent` to method `processEvent` of superclass `Component` (the next object in the chain).

Command

Applications often provide users with several 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 could also be a toolbar and/or a popup menu offering the same items. The functionality the application provides is the same in each case—the different interface components for invoking the functionality are provided for the user's convenience. However, the same GUI component instance (e.g., `JButton`) cannot be used for menus and toolbars and popup 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 specify the desired functionality (e.g., copying text) once in a reusable object; that functionality can
then be added to a menu, toolbar, popup menu or other mechanisms. This design pattern is called Command because it defines a user command, or instruction. This pattern allows a designer to encapsulate a command, so that the command may be used among several objects.

**Observer**

Suppose we want to design a program for viewing bank account information. This system includes class `BankStatementData` to store data pertaining to bank statements, and classes `TextDisplay`, `BarGraphDisplay` and `PieChartDisplay` to display the data. Figure 13.23 shows the design for our system. Class `TextDisplay` displays the data in text format, class `BarGraphDisplay` displays the data in bar-graph format and class `PieChartDisplay` displays the data as a pie chart. We want to design the system so that the `BankStatementData` object notifies the objects displaying the data of a change in the data. We also want to design the system to loosen coupling—the degree to which classes depend on each other in a system.

**Software Engineering Observation 13.9**

Loosely-coupled classes are easier to reuse and modify than are tightly-coupled classes, which depend heavily on each other. A modification in a class in a tightly-coupled system usually results in modifying other classes in that system. A modification to one of a group of loosely-coupled classes would require little or no modification to the other classes in the group.

The Observer design pattern is appropriate for systems like that of Fig. 13.23. This pattern promotes loose coupling between a subject object and observer objects—a subject notifies the observers when the subject changes state. When notified by the subject, the observers change in response to the change in the subject. In our example, the `BankStatementData` object is the subject, and the objects displaying the data are the observers. A subject can notify several observers; therefore, the subject contains a one-to-many relationship with the observers.

2. This approach is the basis for the Model-View-Controller architecture pattern, discussed in Sections 13.17 and 17.11.
The Java API contains classes that use the Observer design pattern. Class `java.util.Observable` represents a subject. Class `Observable` provides method `addObserver`, which takes a `java.util.Observer` argument. Interface `Observer` allows the `Observable` object to notify the `Observer` when the `Observable` objects changes state. 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. If a developer changes the way in which a particular `Observer` responds to changes in the `Observable` object, the developer does not need to change the `Observable` object. The `Observable` object interacts with its `Observers` only through interface `Observer`, which enables the loose coupling.

The optional elevator simulation case study in the “Thinking About Objects” sections uses the Observer design pattern to allow the `ElevatorModel` object (the subject) to notify the `ElevatorView` object (the observer) of changes in the `ElevatorModel`. The simulation does not use class `Observable` and interface `Observer` from the Java library—rather, it uses a custom interface `ElevatorModelListener` that provides functionality similar to that of interface `Observer`.

The Swing GUI components use the Observer design pattern. GUI components collaborate with their listeners to respond to user interactions. For example, an `ActionListener` observes state changes in a `JButton` (the subject) by registering to handle that `JButton`’s events. When pressed by the user, the `JButton` notifies its `ActionListener` objects (the observers) that the `JButton`’s state has changed (i.e., the `JButton` has been pressed).

**Strategy**

The *Strategy design pattern* is similar to the State design pattern (discussed in Section 9.24.3). We mentioned that the State design pattern contains a state object, which encapsulates the state of a context object. The Strategy design pattern contains a *strategy object*, which is analogous to the State design pattern’s state object. The key difference between a state object and a strategy object is that the strategy object encapsulates an *algorithm* rather than state information.

For example, `java.awt.Container` components implement the Strategy design pattern using `LayoutManager`s (discussed in Section 12.14) as strategy objects. In package `java.awt`, classes `FlowLayout`, `BorderLayout` and `GridLayout` implement interface `LayoutManager`. Each class uses method `addLayoutComponent` to add GUI components to a `Container` object—however, each method uses a different algorithm to display these GUI components: a `FlowLayout` displays components in a left-to-right sequence; a `BorderLayout` displays components in five regions; and a `GridLayout` displays components in row-column format.

Class `Container` contains a reference to a `LayoutManager` object (the strategy object). Because an interface reference (i.e., the reference to the `LayoutManager` object) can hold references to objects of classes that implement that interface (i.e., the `FlowLayout`, `BorderLayout` or `GridLayout` objects), the `LayoutManager` object can reference a `FlowLayout`, `BorderLayout` or `GridLayout` at any one time. Class `Container` can change this reference through method `setLayout` to select different layouts at run time.

Class `FlowLayoutDemo` (Fig. 12.24) demonstrates the application of the Strategy pattern—line 16 declares a new `FlowLayout` object and line 19 invokes the Con-
container object’s method `setLayout` to assign the `FlowLayout` object to the `Container` object. In this example, the `FlowLayout` provides the strategy for laying out the components.

**Template Method**

The `Template Method design pattern` also deals with algorithms. The `Strategy` design pattern allows several objects to contain distinct algorithms. However, the `Template Method` design pattern requires all objects to share a single algorithm defined by a superclass.

For example, consider the design of Fig. 13.23, which we mentioned in the Observer design pattern discussion. Objects of classes `TextDisplay`, `BarGraphDisplay` and `PieChartDisplay` use the same basic algorithm for acquiring and displaying the data—get all statements from the `BankStatementData` object, parse the statements then display the statements. The Template Method design pattern allows us to create an abstract superclass called `BankStatementDisplay` that provides the central algorithm for displaying the data. In this example, the algorithm comprises abstract methods `getData`, `parseData` and `displayData` comprise the algorithm. Classes `TextDisplay`, `BarGraphDisplay` and `PieChartDisplay` extend class `BankStatementDisplay` to inherit the algorithm, so each object can use the same algorithm. Each `BankStatementDisplay` subclass then overrides each method in a way specific to that subclass, because each class implements the algorithm differently from one another. For example, classes `TextDisplay`, `BarGraphDisplay` and `PieChartDisplay` might get and parse the data identically, but each class displays that data differently.

The Template Method design pattern allows us to extend the algorithm to other `BankStatementDisplay` subclasses—e.g., we could create classes, such as `LineGraphDisplay` or class `3DimensionalDisplay`, that use the same algorithm inherited from class `BankStatementDisplay`.

**13.18.4 Conclusion**

In this “Discovering Design Patterns” section, we discussed how Swing components take advantage of design patterns and how developers can integrate design patterns with GUI applications in Java. In “Discovering Design Patterns” Section 15.13, we discuss concurrency design patterns, which are particularly useful for developing multithreaded systems.

**SUMMARY**

- `JTextAreas` provide an area for manipulating multiple lines of text. Like class `JTextField`, class `JTextArea` inherits from `JTextComponent`.
- An external event (i.e., an event generated by a different GUI component) normally indicates when the text in a `JTextArea` should be processed.
- Scrollbars are provided for a `JTextArea` by attaching it to a `JScrollPane` object.
- Method `getSelectedText` returns the selected text from a `JTextArea`. Text is selected by dragging the mouse over the desired text to highlight it.
- Method `setText` sets the text in a `JTextArea`.
- To provide automatic word wrap in a `JTextArea`, attach it to a `JScrollPane` with horizontal scrollbar policy `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`. 
• The horizontal and vertical scrollbar policies for a JScrollPane are set when a JScrollPane is constructed or with methods setHorizontalScrollBarPolicy and setVerticalScrollBarPolicy of class JScrollPane.

• A JPanel can be used as a dedicated drawing area that can receive mouse events and is often extended to create new GUI components.

• Swing components that inherit from class JComponent contain method paintComponent, which helps them draw properly in the context of a Swing GUI. JComponent method paintComponent should be overridden to call to the superclass version of paintComponent as the first statement in its body.

• Classes JFrame and JApplet are not subclasses of JComponent; therefore, they do not contain method paintComponent (they have method paint).

• Calling repaint for a Swing GUI component indicates that the component should be painted as soon as possible. The background of the GUI component is cleared only if the component is opaque. Most Swing components are transparent by default. JComponent method setOpaque can be passed a boolean argument indicating whether the component is opaque (true) or transparent (false). The GUI components of package java.awt are different from Swing components in that repaint results in a call to Component method update (which clears the component’s background) and update calls method paint (rather than paintComponent).

• Method setTitle displays a String in a window’s title bar.

• Drawing on any GUI component is performed with coordinates that are measured from the upper-left corner (0, 0) of that GUI component.

• Layout managers often use a GUI component’s getPreferredSize method to determine the preferred width and height of a component when laying out that component as part of a GUI. If a new component has a preferred width and height, it should override method getPreferredSize to return that width and height as an object of class Dimension (package java.awt).

• The default size of a JPanel object is 0 pixels wide and 0 pixels tall.

• A mouse drag operation begins with a mouse-pressed event. All subsequent mouse drag events (for which mouseDragged will be called) are sent to the GUI component that received the original mouse-pressed event.

• JSliders enable the user to select from a range of integer values. JSliders can display major tick marks, minor tick marks and labels for the tick marks. They also support snap-to ticks, where positioning the thumb between two tick marks causes the thumb to snap to the closest tick mark.

• Most Swing GUI components support user interactions through both the mouse and the keyboard.

• If a JSlider has the focus, the left arrow key and right arrow key cause the thumb of the JSlider to decrease or increase by 1. The down arrow key and up arrow key also cause the thumb of the JSlider to decrease or increase by 1, respectively. The PgDn key (page down) and PgUp key (page up) cause the thumb of the JSlider to decrease or increase by block increments of one-tenth of the range of values, respectively. The Home key moves the thumb to the minimum value of the JSlider and the End key moves the thumb to the maximum value of the JSlider.

• JSliders have either a horizontal orientation or a vertical orientation. For a horizontal JSlider, the minimum value is at the extreme left and the maximum value is at the extreme right of the JSlider. For a vertical JSlider, the minimum value is at the extreme bottom and the maximum value is at the extreme top of the JSlider. The relative position of the thumb indicates the current value of the JSlider.

• Method setMajorTickSpacing of class JSlider sets the spacing for tick marks on a JSlider. Method setPaintTicks with a true argument indicates that the tick marks should be displayed.
• **JSlider**s generate **ChangeEvent**s (package **javax.swing.event**) when the user interacts with a **JSlider**. A **ChangeListener** (package **javax.swing.event**) defines method **stateChanged** that can respond to **ChangeEvent**s.

• Method **getValue** of class **JSlider** returns the current thumb position.

• A **JFrame** is a window with a title bar and a border. Class **JFrame** is a subclass of **java.awt.Frame** (which is a subclass of **java.awt.Window**).

• Class **JFrame** supports three operations when the user closes the window. By default, a **JFrame** is hidden when the user closes a window. This can be controlled with **JFrame** method **setDefaultCloseOperation**. Interface **WindowConstants** (package **javax.swing**) defines three constants for use with this method—**DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE** and **HIDE_ON_CLOSE** (the default).

• By default, a window is not displayed on the screen until its **setVisible** method is called with **true** as an argument. A window can also be displayed by calling its **show** method.

• A window’s size should be set with a call to method **setSize**. The position of a window when it appears on the screen is specified with method **setLocation**.

• All windows generate window events when the user manipulates the window. Event listeners are registered for window events with method **addWindowListener** of class **Window**. The **WindowListener** interface provides seven methods for handling window events—**windowActivated** (called when the window is made active by clicking the window), **windowClosed** (called after the window is closed), **windowClosing** (called when the user initiates closing of the window), **windowDeactivated** (called when another window is made active), **windowIconified** (called when the user minimizes a window), **windowDeiconified** (called when a window is restored from being minimized) and **windowOpened** (called when a window is first displayed on the screen).

• The command-line arguments are automatically passed to **main** as the array of **Strings** called **args**. The first argument after the application class name is the first **String** in the array **args**, and the length of the array is the total number of command-line arguments.

• **Menus** are an integral part of GUIs. Menus allow the user to perform actions without unnecessarily “cluttering” a graphical user interface with extra GUI components.

• In Swing GUIs, menus can be attached only to objects of the classes that provide method **setJMenuBar**. Two such classes are **JFrame** and **JApplet**.

• The classes used to define menus are **JMenuBar**, ** JMenuItem**, **JMenu**, **JCheckBoxMenuItem** and class **JRadioButtonMenuItem**.

• A **JMenuBar** is a container for menus.

• A **JMenuItem** is a GUI component inside a menu that, when selected, causes an action to be performed. A **JMenuItem** can be used to initiate an action or it can be a **submenu** that provides more menu items from which the user can select.

• A **JMenu** contains menu items and can be added to a **JMenuBar** or to other **JMenus** as submenus. When a menu is clicked, the menu expands to show its list of menu items.

• When a **JCheckBoxMenuItem** is selected, a check appears to the left of the menu item. When the **JCheckBoxMenuItem** is selected again, the check to the left of the previously selected menu item is removed.

• When multiple **JRadioButtonMenuItems** are maintained as part of a **ButtonGroup**, only one item in the group can be selected at a given time. When a **JRadioButtonMenuItem** is selected, a filled circle appears to the left of the menu item. When another **JRadioButtonMenuItem** is selected, the filled circle to the left of the previously selected menu item is removed.

• **JFrame** method **setJMenuBar** attaches a menu bar to a **JFrame**.
AbstractButton method `setMnemonic` (inherited into class JMenu) specifies the mnemonic for an AbstractButton object. Pressing the Alt key and the mnemonic performs the AbstractButton’s action (in the case of a menu, it opens the menu).

- Mnemonic characters are normally displayed with an underline.
- Dialog boxes can be either modal or modeless. A modal dialog box does not allow any other window in the application to be accessed until the dialog box is dismissed. A modeless dialog box allows other windows to be accessed while the dialog is displayed. By default, the dialogs displayed with class JOptionPane are modal dialogs. Class JDialog can be used to create your own modeless or modal dialogs.

- JMenu method `addSeparator` adds a separator line to a menu.
- Context-sensitive popup menus are created with class JPopupMenu. These menus provide options that are specific to the component for which the popup-trigger event was generated. On most systems, the popup-trigger event occurs when the user presses and releases the right mouse button.

- MouseEvent method `isPopupTrigger` returns `true` if the popup-trigger event occurred.
- Method `show` of class JPopupMenu displays a JPopupMenu. The first argument to method show specifies the origin component, whose position helps determine where the JPopupMenu will appear on the screen. The last two arguments are the x and y coordinates from the origin component’s upper-left corner at which the JPopupMenu should appear.

- UIManager contains a public static inner class called LookAndFeelInfo that is used to maintain information about a look-and-feel.

- UIManager static method `getInstalledLookAndFeels` gets an array of UIManager.LookAndFeelInfo objects that describe the installed look-and-feels.

- UIManager static method `setLookAndFeel` changes the look-and-feel.

- SwingUtilities static method `updateComponentTreeUI` changes the look-and-feel of every component attached to its Component argument to the new look-and-feel.

- Many of today’s applications use a multiple document interface (MDI) [i.e., a main window (often called the parent window) containing other windows (often called child windows)] to manage several open documents that are being processed in parallel.

- Swing’s JDesktopPane and JInternalFrame classes provide support for creating multiple document interfaces.

- BoxLayout is a layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class Box defines a container with BoxLayout as its default layout manager and provides static methods to create a Box with a horizontal or vertical BoxLayout.

- CardLayout is a layout manager that stacks components like a deck of cards. Each container in the stack can use any layout manager. Only the container at the “top” of the deck is visible.

- GridBagLayout is a layout manager similar to GridLayout. Unlike with GridLayout, each component size can vary, and components can be added in any order.

- Box static method `createHorizontalBox` returns a Box container with a horizontal BoxLayout. Box static method `createVerticalBox` of class Box returns a Box container with a vertical BoxLayout.

- Box static method `createVerticalStrut` adds a vertical strut to a container. A vertical strut is an invisible GUI component that has a fixed pixel height and is used to guarantee a fixed amount of space between GUI components. Class Box also defines method `createHorizontalStrut` for horizontal BoxLayouts.

- Box static method `createHorizontalGlue` adds horizontal glue to a container. Horizontal glue is an invisible GUI component that can be used between fixed-size GUI components to
occupy additional space. Class **Box** also defines method `createVerticalGlue` for vertical `BoxLayout`

- **Box static** method `createRigidArea` adds a rigid area to a container. A rigid area is an invisible GUI component that always has a fixed pixel width and height.
- The **BoxLayout** constructor receives a reference to the container for which it controls the layout and a constant indicating whether the layout is horizontal (**BoxLayout.X_AXIS**) or vertical (**BoxLayout.Y_AXIS**).
- **CardLayout** methods `first`, `previous`, `next` and `last` are used to display a particular card. Method `first` displays the first card. Method `previous` displays the previous card. Method `next` displays the next card. Method `last` displays the last card.
- To use **GridLayout**, a **GridLayout** object must be used to specify how a component is placed in a **GridLayout**.
- Method `setConstraints` of class **GridLayout** takes a **Component** argument and a **GridBagConstraints** argument and sets the constraints of the **Component**.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Method/Field</th>
<th>Class/Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addSeparator</code></td>
<td><strong>JMenu</strong> class</td>
</tr>
<tr>
<td><code>addWindowListener</code></td>
<td><strong>GridBagConstraints</strong> class</td>
</tr>
<tr>
<td><code>Box</code></td>
<td><strong>JMenu</strong> class</td>
</tr>
<tr>
<td><code>BoxLayout</code> layout manager</td>
<td><strong>GridBagConstraints</strong> class</td>
</tr>
<tr>
<td><code>BoxLayout.X_AXIS</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>BoxLayout.Y_AXIS</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>CardLayout</code> layout manager</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>ChangeEvent</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>ChangeListener</code> interface</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>child window</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>command-line arguments</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>context-sensitive popup menu</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createHorizontalBox</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createHorizontalGlue</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createHorizontalStrut</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createRigidArea</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createVerticalBox</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createVerticalGlue</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>createVerticalStrut</code> method of <strong>Box</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>dedicated drawing area</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>Dimension</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>dispose</code> method of class <strong>Window</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>external event</code></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>fill</code> variable of <strong>GridBagConstraints</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>first</code> method of <strong>CardLayout</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>getClassName</code> method of <strong>JMenuItem</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>getInstalledLookAndFeels</code> method of <strong>Component</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>getMinimumSize</code> method of <strong>Component</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>getPreferredSize</code> method of <strong>Component</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>getSelectedText</code> method</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>getValue</code> method of class <strong>JSlider</strong></td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JCheckModeMenuItem</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JDesktopPane</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JInternalFrame</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JMenu</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JMenuBar</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JMenuBase</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JMenuItem</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JPopupMenu</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JRadioButtonMenuItem</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JSlider</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JTextArea</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
<tr>
<td><code>JTextComponent</code> class</td>
<td><strong>GridLayout</strong> layout manager</td>
</tr>
</tbody>
</table>
SELF-REVIEW EXERCISES

13.1 Fill in the blanks in each of the following statements:
   a) The ________ class is used to create a menu object.
   b) The ________ method places a separator bar in a menu.
   c) Passing false to a TextArea’s ________ method prevents its text from being modified by the user.
   d) JSlider events are handled by the ________ method of interface ________.
   e) The GridBagConstraints instance variable ________ is set to CENTER by default.

13.2 State whether each of the following is true or false. If false, explain why.
   a) When the programmer creates a JFrame, a minimum of one menu must be created and added to the JFrame.
   b) The variable fill belongs to the GridBagLayout class.
   c) JFrame and applets cannot be used together in the same program.
   d) The top-left corner of a JFrame or applet has a coordinate of (0, 0).
   e) A JTextArea’s text is always read-only.
   f) Class JTextArea is a direct subclass of class Component.
   g) The default layout for a Box is BoxLayout.

13.3 Find the error(s) in each of the following and explain how to correct the error(s).
   a) JMenuBar b;
   b) mySlider = JSlider( 1000, 222, 100, 450 );
c) `gbc.fill = GridBagConstraints.NORTHWEST;`  // set fill

d) `public void paintcomponent( Graphics g )`  
   
   `{  
       g.drawString( "HELLO", 50, 50 );  
   }

e) `// create a JFrame and display it`  
   
   `JFrame f = new JFrame( "A Window" );`  
   `f.setVisible( true );`  

**ANSWERS TO SELF-REVIEW EXERCISES**

13.1  a) `JMenu`.  b) `addSeparator`.  c) `setEditable`.  d) `stateChanged`, `ChangeListener`.  e) `anchor`.

13.2  a) False. A `JFrame` does not require any menus.
   
   b) False. The variable `fill` belongs to the `GridBagConstraints` class.
   
   c) False. They can be used together.
   
   d) True.
   
   e) False. `JTextArea` are editable by default.
   
   f) False. `JTextArea` derives from class `JTextComponent`.
   
   g) True.

13.3  a) `JMenubar` should be `JMenuBar`.
   
   b) The first argument to the constructor should be either `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL` and the `new` operator must be used after the `=` operator.
   
   c) The constant should be either `BOTH`, `HORIZONTAL`, `VERTICAL` or `NONE`.
   
   d) `paintcomponent` should be `paintComponent` and the method should call `super.paintComponent( g )` as its first statement.
   
   e) The `JFrame`’s `setSize` method must also be called to determine the size of the window.

**EXERCISES**

13.4  Fill in the blanks in each of the following statements:
   
   a) A dedicated drawing area can be defined as a subclass of _________.
   
   b) A `JMenuItem` that is a `JMenu` is called a _________.
   
   c) Both `JTextField`s and `JTextArea`s inherit directly from class _________.
   
   d) The _________ method attaches a `JMenuBar` to a `JFrame`.
   
   e) Container class _________ has a default `BoxLayout`.
   
   f) A _________ manages a set of child windows defined with class `JInternalFrame`.

13.5  State whether each of the following is `true` or `false`. If `false`, explain why.
   
   a) Menus require a `JMenuBar` object so they can be attached to a `JFrame`.
   
   b) A `JPanel` object is capable of receiving mouse events.
   
   c) `CardLayout` is the default layout manager for a `JFrame`.
   
   d) Method `setEditable` is a `JTextComponent` method.
   
   e) The `GridBagLayout` layout manager implements `LayoutManager`.
   
   f) `JPanel` objects are containers to which other GUI components can be attached.
   
   g) Class `JFrame` inherits directly from class `Container`.
   
   h) `JApplets` can contain menus.

13.6  Find the error(s) in each of the following. Explain how to correct the error(s).
   
   a) `x.add( new JMenuItem( "Submenu Color" ) );`  // create submenu
b) `container.setLayout( m = new GridBagLayout() );`

c) `String s = JTextArea.getText();`

13.7 Write a program that displays a circle of random size and calculates and displays the area, radius, diameter and circumference. Use the following equations: \( \text{diameter} = 2 \times \text{radius} \), \( \text{area} = \pi \times \text{radius}^2 \), \( \text{circumference} = 2 \times \pi \times \text{radius} \). Use the constant `Math.PI` for pi (\( \pi \)). All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

13.8 Enhance the program of Exercise 13.7 by allowing the user to alter the radius with a `JSlider`. The program should work for all radii in the range from 100 to 200. As the radius changes, the diameter, area and circumference should be updated and displayed. The initial radius should be 150. Use the equations of Exercise 13.7. All drawing should be done on a subclass of `JPanel`, and the results of the calculations should be displayed in a read-only `JTextArea`.

13.9 Explore the effects of varying the `weightx` and `weighty` values of the program of Fig. 13.20. What happens when a component has a nonzero weight, but is not allowed to fill the whole area (i.e., the `fill` value is not `BOTH`)?

13.10 Write a program that uses the `paintComponent` method to draw the current value of a `JSlider` on a subclass of `JPanel`. In addition, provide a `JTextField` where a specific value can be entered. The `JTextField` should display the current value of the `JSlider` at all times. A `JLabel` should be used to identify the `JTextField`. The `JSlider` methods `setValue` and `getValue` should be used. [Note: The `setValue` method is a `public` method that does not return a value and takes one integer argument—the `JSlider` value, which determines the position of the thumb.]

13.11 Modify the program of Fig. 13.16 to use a single `JComboBox` instead of the four separate `JButtons`. Each “card” should not be modified.

13.12 Modify the program of Fig. 13.16 by adding a minimum of two new “cards” to the deck.

13.13 Define a subclass of `JPanel` called `MyColorChooser` that provides three `JSlider` objects and three `JTextField` objects. Each `JSlider` represents the values from 0 to 255 for the red, green and blue parts of a color. Use the red, green and blue values as the arguments to the `Color` constructor to create a new `Color` object. Display the current value of each `JSlider` in the corresponding `JTextField`. When the user changes the value of the `JSlider`, the `JTextField` should be changed accordingly. Define class `MyColorChooser` so it can be reused in other applications or applets. Use your new GUI component as part of an applet that displays the current `Color` value by drawing a filled rectangle.

13.14 Modify the `MyColorChooser` class of Exercise 13.13 to allow the user to type an integer value into a `JTextField` to set the red, green or blue value. When the user presses `Enter` in the `JTextField`, the corresponding `JSlider` should be set to the appropriate value.

13.15 Modify the applet of Exercise 13.14 to draw the current color as a rectangle on an instance of a subclass of `JPanel` called `DrawPanel`. Class `DrawPanel` should provide its own `paintComponent` method to draw the rectangle and should provide `set` methods to set the red, green and blue values for the current color. When any `set` method is invoked for the class `DrawPanel`, the object should automatically repain itself.

13.16 Modify the applet of Exercise 13.15 to allow the user to drag the mouse across the `DrawPanel` to draw a shape in the current color. Enable the user to choose what shape to draw.

13.17 Modify the program of Exercise 13.16 to enable the program to run as an application. The existing applet’s code should be modified only by adding a `main` method to launch the application in its own `JFrame`. Provide the user with the ability to terminate the application by clicking the close
box on the window that is displayed and by selecting **Exit** from a **File** menu. Use the techniques shown in Fig. 13.9.

**13.18 (Complete Drawing Application)** Using the techniques developed in Exercise 12.27–Exercise 12.33 and Exercise 13.13–Exercise 13.17, create a complete drawing program that can execute as both an applet and an application. The program should use the GUI components of Chapter 12 and Chapter 13 to enable the user to select the shape, color and fill characteristics. Each shape should be stored in an array of **MyShape** objects, where **MyShape** is the superclass in your hierarchy of shape classes (see Exercise 9.28 and Exercise 9.29). Use a **JDesktopPane** and **JInternalFrame**s to allow the user to create multiple separate drawings in separate child windows. Create the user interface as a separate child window containing all the GUI components that allow the user to determine the characteristics of the shape to be drawn. The user can then click in any **JInternalFrame** to draw the shape.

**13.19** A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and “time-and-a-half,” i.e., 1.5 times their hourly wage, for overtime hours worked), commission workers (who receive $250 plus 5.7% of their gross weekly sales) or pieceworkers (who receive a fixed amount of money per item for each of the items they produce—each pieceworker in this company works on only one type of item). Write an application to compute the weekly pay for each employee. Each type of employee has its own pay code: Managers have paycode 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a **switch** to compute each employee’s pay based on that employee’s paycode. Use a **CardLayout** to display the appropriate GUI components that allow the user to enter the facts your program needs to calculate each employee’s pay based on that employee’s paycode.
Objectives

• To understand exception and error handling.
• To be able to use `try` blocks to delineate code in which an exception may occur.
• To be able to `throw` exceptions.
• To use `catch` blocks to specify exception handlers.
• To use the `finally` block to release resources.
• To understand the Java exception hierarchy.
• To create programmer-defined exceptions.

*It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.*
Franklin Delano Roosevelt

*O! throw away the worser part of it, And live the purer with the other half.*
William Shakespeare

*If they’re running and they don’t look where they’re going I have to come out from somewhere and catch them.*
Jerome David Salinger

*And oftentimes excusing of a fault Doth make the fault the worse by the excuse.*
William Shakespeare

*I never forget a face, but in your case I’ll make an exception.*
Groucho (Julius Henry) Marx
14.1 Introduction

In this chapter, we introduce exception handling. An exception is an indication that a problem occurred during the program’s execution. The extensibility of Java can increase the number and types of errors that can occur. Every new class can add its own error possibilities. The features presented here enable programmers to write clearer, more robust, more fault-tolerant programs. We also consider when exception handling should not be used.

The style and details of exception handling in Java as presented in this chapter are based in part on the work of Andrew Koenig and Bjarne Stroustrup as presented in their paper, “Exception Handling for C++ (revised),” published in the Proceedings of the USENIX Conference held in San Francisco in April 1990. Their work forms the basis of C++ exception handling. Java’s designers chose to implement an exception handling mechanism similar to that used in C++.

Error-handling code varies in nature and quantity among software systems depending on the application and whether the software is a product for release. Products tend to contain far more error-handling code than does “casual” software.

There are many popular means for dealing with errors. Most commonly, error-handling code is interspersed throughout a system’s code. Errors are dealt with at the places in the code where the errors can occur. The advantage to this approach is that a programmer reading code can see the error processing in the immediate vicinity of the code and determine if the proper error checking has been implemented.

The problem with this scheme is that the code can become “polluted” with the error processing. It becomes more difficult for a programmer concerned with the application itself to read the code and determine if it is functioning correctly. This can make understanding and maintaining the application difficult.
Some common examples of exceptions are an out-of-bounds array subscript, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid method parameters and memory exhaustion.

**Good Programming Practice 14.1**

*Using Java exception handling enables the programmer to remove the error-handling code from the “main line” of the program’s execution. This improves program clarity and enhances modifiability.*

Exception handling is provided to enable programs to catch and handle errors rather than letting them occur and suffering the consequences. Exception handling is designed for dealing with synchronous errors such as an attempt to divide by zero (that occurs as the program executes the divide instruction). Exception handling is not designed to deal with asynchronous events such as disk I/O completions, network message arrivals, mouse clicks, keystrokes and the like; these are best handled through other means, such as Java event listeners.

Java exception handling enables a program to catch all exceptions, all exceptions of a certain type or all exceptions of related types. This flexibility makes programs more robust by reducing the likelihood that programs will not process problems during program execution.

Exception handling is used in situations in which the system can recover from the malfunction that caused the exception. The recovery procedure is called an exception handler. The exception handler can be defined in the method that may cause an exception or in a calling method.

**Software Engineering Observation 14.1**

*Use exceptions for malfunctions that must be processed in a different method from where they are detected. Use conventional error-handling techniques for local error processing in which a method is able to deal with its own exceptions.*

Exception handling is designed to process exceptional conditions—problems that do not happen frequently, but can happen. It is possible that the exception-handling code may not be optimized to the same performance levels as other language elements.

**Performance Tip 14.1**

*When an exception does not occur, little or no overhead is imposed by the presence of exception handling code. When exceptions happen, they do incur execution-time overhead.*

**Testing and Debugging Tip 14.1**

*Exception handling helps improve a program’s fault tolerance.*

**Good Programming Practice 14.2**

*Using Java’s standardized exception handling rather than having programmers use a diversity of “home-grown” techniques improves program clarity on large projects.*

We will see that exceptions are objects of classes derived from superclass **Exception**. We will show how to deal with “uncaught” exceptions. We will consider how unexpected exceptions are handled by Java. We will show how related exception types can be represented by exception subclasses that are derived from a common exception superclass.

Exception handling can be viewed as another means of returning control from a method or exiting a block of code. Normally, when an exception occurs, the exception is
handled by a caller of the method generating the exception, by a caller of that caller, or however far back in the call stack it becomes necessary to go to find a handler for that exception.

Software Engineering Observation 14.2
Exception handling is particularly well-suited to systems of separately developed components. Such systems are typical of real-world software. Exception handling makes it easier to combine the components and have them work together effectively.

Software Engineering Observation 14.3
With other programming languages that do not support exception handling, programmers often delay writing error-processing code, and sometimes programmers simply forget to include it. This results in less-robust, and thus inferior, software products. Java forces the programmer to deal with exception handling from the inception of a project. Still, the programmer must put considerable effort into incorporating an exception-handling strategy into software projects.

Software Engineering Observation 14.4
It is best to incorporate your exception-handling strategy into a system from the inception of the design process. It is difficult to add effective exception handling after a system has been implemented.

14.2 When Exception Handling Should Be Used
Exception handling should be used

- to process exceptional situations where a method is unable to complete its task for reasons it cannot control,
- to process exceptions from program components that are not geared to handling those exceptions directly, or
- on large projects to handle exceptions in a uniform manner project wide.

Software Engineering Observation 14.5
The client of a library class will likely have unique error processing in mind for an exception generated in the library class. It is unlikely that a library class will perform error processing that would meet the unique needs of all clients. Exceptions are an appropriate means for dealing with errors produced by library classes.

14.3 Other Error-Handling Techniques
We have presented various ways of dealing with exceptional situations prior to this chapter. A program can ignore some exception types. This can be devastating for software products released to the general public, or for special-purpose software needed for mission-critical situations. But for software developed for your own purposes, it is common to ignore many kinds of errors. A program could be directed to abort upon encountering an exceptional situation. This prevents a program from running to completion and producing incorrect results. For many types of errors this is a good strategy. Such a strategy is inappropriate for mission-critical applications. Resource issues also are important here. If a program obtains a resource, the program should return that resource before program termination.
Common Programming Error 14.1
Aborting a program could leave a resource in a state in which other programs would not be able to acquire the resource; hence, we would have a so-called “resource leak.”

Good Programming Practice 14.3
If your method is capable of handling a given type of exception, then handle it rather than passing the exception on to other regions of your program. This makes programs clearer.

Performance Tip 14.2
If an error can be processed locally instead of throwing an exception, do so. This will improve program execution speed. Exception handling is slow compared to local processing.

14.4 Basics of Java Exception Handling
In this section, we overview the Java exception-handling process. Throughout the chapter, we present detailed discussions of the steps discussed here.

Java exception handling is geared to situations in which the method that detects an error is unable to deal with it. Such a method will throw an exception. There is no guarantee that there will be “anything out there” (i.e., an exception handler—code that executes when the program detects an exception) to process that kind of exception. If there is, the exception will be caught and handled. The following Testing and Debugging Tip describes what happens if no appropriate exception handler can be found.

Testing and Debugging Tip 14.2
All Java applets and certain Java applications are GUI-based. Some Java applications are not GUI-based; these are often called command-line applications (or console applications). When an exception is not caught in a command-line application, the program terminates (i.e., Java exits) after the default exception handler runs. When an exception is not caught in an applet or a GUI-based application, the GUI remains displayed and the user can continue using the applet or application even after the default exception handler runs. However, the program may be in an inconsistent state and may produce incorrect results.

The programmer encloses in a try block the code that may generate an exception and any code that should not execute if an exception occurs. The try block is followed by zero or more catch blocks. Each catch block specifies the type of exception it can catch and contains an exception handler. After the last catch block, an optional finally block provides code that always executes, regardless of whether an exception occurs. As we will see, the finally block is an ideal location for code that releases resources to prevent “resource leaks.” The try block must be followed by a catch block or a finally block.

When a method throws an exception, program control leaves the try block and continues execution at the first catch block. The program searches the catch blocks in order looking for an appropriate handler. (We will soon discuss what makes a handler “appropriate”.) If the type of the thrown exception matches the parameter type in one of the catch blocks, the code for that catch block executes. If a try block completes successfully without throwing any exceptions, the program skips the exception handlers for that block and resumes execution after the last catch block. If a finally block appears after the last catch block, the finally block executes regardless of whether an exception occurs.

In a method definition, a throws clause specifies the exceptions the method throws. This clause appears after the parameter list and before the method body. The clause con-
tains a comma-separated list of potential exceptions the method will throw if a problem occurs while the method executes. Such exceptions may be thrown by statements in the method’s body, or they may be thrown by methods called in the body. The point at which the `throw` occurs is called the *throw point*.

When an exception occurs, the block in which the exception occurred expires (terminates)—program control cannot return directly to the throw point. Java uses the *termination model of exception handling* rather than the *resumption model of exception handling*. In the resumption model, control would return to the point at which the exception occurred and resume execution.

When an exception occurs, it is possible to communicate information to the exception handler from the vicinity in which the exception occurred. That information is the type of thrown exception object or information harvested from the vicinity in which the exception occurred and placed into the thrown object.

### 14.5 `try` Blocks

An exception that occurs in a `try` block normally is caught by an exception handler specified by a `catch` block immediately following that `try` block as in

```java
try {
    statements that may throw an exception
} catch( ExceptionType exceptionReference ) {
    statements to process an exception
}
```

A `try` block can be followed by zero or more `catch` blocks. If a `try` block executes and no exceptions are thrown, all the exception handlers are skipped and control resumes with the first statement after the last exception handler. If a `finally` block (presented in Section 14.13) follows the last `catch` block, the code in the `finally` block executes regardless of whether an exception is thrown. Note that an exception handler cannot access objects defined in the corresponding `try` block, because the `try` block expires before the handler begins executing.

**Common Programming Error 14.2**

It is a syntax error to separate with other code the `catch` handlers that correspond to a particular `try` block.

### 14.6 Throwing an Exception

The `throw` statement is executed to indicate that an exception has occurred (i.e., a method could not complete successfully). This is called *throwing an exception*. A `throw` statement specifies an object to be thrown. The operand of a `throw` can be of any class derived from class `Throwable` (package `java.lang`). The two immediate subclasses of class `Throwable` are `Exception` and `Error`. `Errors` are particularly serious system problems that generally should not be caught. `Exceptions` are caused by problems that should be caught and processed during program execution to make a program more robust. If the operand of the `throw` is an object of class `Exception`, it is called an *exception object*. 
Testing and Debugging Tip 14.3

When `toString` is invoked on any `Throwable` object, its resulting `String` includes the descriptive `String` that was supplied to the constructor, or simply the class name if no `String` was supplied.

Testing and Debugging Tip 14.4

An object can be thrown without containing information about the problem that occurred. In this case, simple knowledge that an exception of a particular type occurred may provide sufficient information for the handler to process the problem correctly.

When an exception is thrown, control exits the current `try` block and proceeds to an appropriate `catch` handler (if one exists) after that `try` block. It is possible that the `throw` point could be in a deeply nested scope within a `try` block; control will still proceed to the `catch` handler. It is also possible that the `throw` point could be in a deeply nested method call; still, control will proceed to the `catch` handler.

A `try` block may appear to contain no error checking and include no `throw` statements, but methods called from the `try` block may throw exceptions. Also, statements in a `try` block that do not invoke methods may cause exceptions. For example, a statement that performs array subscripting on an array object throws an `ArrayIndexOutOfBoundsException` if the statement specifies an invalid array subscript. Any method call can invoke code that might `throw` an exception or call another method that throws an exception.

14.7 Catching an Exception

Exception handlers are contained in `catch` blocks. Each `catch` block starts with the keyword `catch` followed by parentheses containing a class name (specifying the type of exception to be caught) and a parameter name. The handler can reference the thrown object through this parameter. This is followed by a block delineating the exception-handling code. When a handler catches an exception, the code in the `catch` block executes.

Common Programming Error 14.3

Logic errors can occur if you assume that after an exception is processed, control will return to the first statement after the `throw`. Program control continues with the first statement after the `catch` handlers.

Common Programming Error 14.4

Specifying a comma-separated list of `catch` parameters is a syntax error. A `catch` can have only a single parameter.

Common Programming Error 14.5

It is a syntax error to `catch` the same type in two different `catch` blocks associated with a particular `try` block.

A `catch` that catches a `Throwable` object

```
catch( Throwable throwable )
```
catches all exceptions and errors. Similarly, a `catch` that catches an `Exception` object

```
catch( Exception exception )
```
catches all exceptions. In general, programs do not define exception handlers for type Throwable, because Errors normally should not be caught in a program.

**Common Programming Error 14.6**

*Placing catch(Exception exception) before other catch blocks that catch specific types of exceptions would prevent those blocks from executing; an exception handler that catches type Exception must be placed last in the list of exception handlers following a try block, or a syntax error occurs.*

It is possible that a try block will not have a corresponding catch handler that matches a particular thrown object. This causes the search for a matching catch handler to continue in the next enclosing try block. As this process continues, eventually the program may determine that there is no handler on the execution stack that matches the type of the thrown object. In this case, a non-GUI-based application terminates—applets and GUI-based applications return to their regular event processing. Although applets and GUI-based applications continue to execute, they may execute incorrectly.

**Software Engineering Observation 14.6**

*If you know that a method may throw an exception, include appropriate exception-handling code in your program. This will make your program more robust.*

**Good Programming Practice 14.4**

*Read the online API documentation for a method before using that method in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur.*

**Good Programming Practice 14.5**

*Read the online API documentation for an exception class before writing exception-handling code for that type of exception. The documentation for an exception class typically contains potential reasons that such exceptions may occur during program execution.*

It is possible that several exception handlers will provide an acceptable match to the type of the exception. This can happen for several reasons: There can be a “catch-all” handler catch(Exception exception) that will catch any exception. Also, inheritance relationships enable a subclass object to be caught either by a handler specifying the subclass type, or by handlers specifying the types of any of that class’s superclasses. The first exception handler that matches the exception type executes—all other exception handlers for the corresponding try block are ignored.

**Software Engineering Observation 14.7**

*If several handlers match the type of an exception, and if each of these handles the exception differently, then the order of the handlers will affect the manner in which the exception is handled.*

**Common Programming Error 14.7**

*It is a syntax error if a catch that catches a superclass object is placed before a catch for that class’s subclass types.*

Sometimes a program may process many closely related types of exceptions. Instead of providing separate catch handlers for each, a programmer can provide a single catch handler for a group of exceptions.
What happens when an exception occurs in an exception handler? The try block that noticed the exception expires before the exception handler begins running, so exceptions occurring in an exception handler are processed by catch handlers for an enclosing try block. The enclosing try block watches for errors occurring in the original try block’s catch handlers. An enclosing try block is either a try block that contains a complete try/catch sequence or a try block in a calling method.

Exception handlers can be written a variety of ways. They can rethrow an exception (as we will see in Section 14.9). They can convert one type of exception into another by throwing a different type of exception. They can perform any necessary recovery and resume execution after the last exception handler. They can look at the situation causing the error, remove the cause of the error and retry by calling the original method that caused an exception. They can return a status value to their environment, etc.

It is not possible to return to the throw point by issuing a return statement in a catch handler. Such a return simply returns control to the method that called the method containing the catch block. Again, the throw point is in a block that has expired, so returning to that point via a return statement would not make sense.

Software Engineering Observation 14.8

Another reason not to use exceptions for conventional flow of control is that these “addition-al” exceptions can “get in the way” of genuine error-type exceptions. It becomes more difficult for the programmer to keep track of the larger number of exception cases. Exceptional situations should be rare, not commonplace.

Common Programming Error 14.8

Assuming that an exception thrown from a catch handler will be processed by that handler or any other handler associated with the same try block can lead to logic errors.

14.8 Exception-Handling Example: Divide by Zero

Now let us consider a simple example of exception handling. The application of Fig. 14.1 and Fig. 14.2 uses try, throw and catch to detect, indicate and handle exceptions. The application displays two JTextFields in which the user can type integers. When the user presses the Enter key in the second JTextField, the program calls method actionPerformed to read the two integers from the JTextFields and pass the integers to method quotient, which calculates the quotient of the two values and returns a double result. If the user types 0 in the second JTextField, the program uses an exception to indicate that the user is attempting to divide by zero. Also, if the user types a noninteger value in either JTextField, a NumberFormatException occurs. In prior examples that read numeric values from the user, we simply assumed that the user would input a proper integer value. However, users sometimes make mistakes. This program demonstrates how to catch the NumberFormatException that occurs when a program attempts to convert a String that does not represent an integer value to an int value with Integer method parseInt.

Before we discuss the program, consider the sample executions shown in the five output windows of Fig. 14.2. The first window shows a successful execution. The user typed the values 100 and 7. The third JTextField shows the result of the division performed by method quotient. In the second output window, the user entered the string “hello” in the second JTextField. When the user presses Enter in the second JTextField,
Field, an error-message dialog is displayed indicating that an integer must be entered. In the last two windows, a zero denominator is entered and the program detects the problem, throws an exception and issues an appropriate diagnostic message.

Now let’s discuss the program beginning with class DivideByZeroException of Fig. 14.1. Java can test for division by zero when the values in the division are both integers. If Java discovers an attempt to divide by zero in integer arithmetic, Java throws an ArithmeticException. However, our program performs a floating-point division of two integers by casting the first integer to a double before performing the calculation. Java allows floating-point division by zero. The result is positive or negative infinity. (Classes Float and Double of package java.lang each provide constants that represent these values.) Even though Java allows floating-point division by zero, we would like to use exception handling in this example to indicate to the user of our program that they are attempting to divide by zero.

As we will see, method quotient throws an exception when it receives zero as its second argument. A method can throw an exception of any existing type in the Java API or of a type specific to the program. In this example, we demonstrate defining a new exception type. Actually, we could use class ArithmeticException (package java.lang) with a customized error message in this program.

**Good Programming Practice 14.6**

Associating each type of serious execution-time malfunction with an appropriately named Exception class improves program clarity.

**Software Engineering Observation 14.9**

If possible, use an existing exception type, rather than creating a new class. The Java API contains many exception types that may be suitable for your program.

```java
// Fig. 14.1: DivideByZeroException.java
// Definition of class DivideByZeroException.
// Used to throw an exception when a
// divide-by-zero is attempted.
public class DivideByZeroException extends ArithmeticException {

    // no-argument constructor specifies default error message
    public DivideByZeroException() {
        super( "Attempted to divide by zero" );
    }

    // constructor to allow customized error message
    public DivideByZeroException( String message )
    {
        super( message );
    }

} // end class DivideByZeroException
```

**Fig. 14.1** Exception class DivideByZeroException.
Class **DivideByZeroException** extends class **ArithmeticException**. We chose to extend class **ArithmeticException** because dividing by zero occurs during arithmetic. Like any other class, an exception class can contain instance variables and methods. A typical exception class contains only two constructors—one that takes no arguments and specifies a default exception message and one that receives a customized exception message as a **String**. The default constructor (lines 8–11) specifies the string "**Attempted to divide by zero**" as the exception message that indicates what went wrong. This string is passed to the superclass constructor to initialize the error message associated with the exception object. The other constructor (lines 14–17) passes its argument—a customized error-message string—to the superclass constructor.

**Software Engineering Observation 14.10**

When defining your own exception type, subclass an existing related exception type in the Java API. This requires you to investigate the existing exceptions in the Java API. If the existing types are not appropriate for subclassing, the new exception class should extend **Exception** if the client of your code should be required to handle the exception, or should extend **RuntimeException** if the client of your code should have the option of ignoring the exception.

Now consider the **DivideByZeroTest** application (Fig. 14.2). The application’s constructor (lines 21–51) builds a graphical user interface with three **JLabel**s (all right aligned) and three **JTextField**s and registers the **DivideByZeroTest** object as the **ActionListener** for **JTextField inputField2**.

```java
// Fig. 14.2: DivideByZeroTest.java
// A simple exception handling example.
// Checking for a divide-by-zero-error.

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

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

// Fig. 14.2: DivideByZeroTest.java
public class DivideByZeroTest extends JFrame implements ActionListener {

    private JTextField inputField1, inputField2, outputField;
    private int number1, number2;
    private double result;

    // set up GUI
    public DivideByZeroTest() {
        super( "Demonstrating Exceptions" );
    }
```
// get content pane and set its layout
Container container = getContentPane();
container.setLayout( new GridLayout( 3, 2 ) );

// set up label and inputField1
container.add( 
    new JLabel( "Enter numerator ", SwingConstants.RIGHT ) );
inputField1 = new JTextField( 10 );
container.add( inputField1 );

// set up label and inputField2; register listener
container.add( 
    new JLabel( "Enter denominator and press Enter ", 
        SwingConstants.RIGHT ) );
inputField2 = new JTextField( 10 );
container.add( inputField2 );
inputField2.addActionListener( this );

// set up label and outputField
container.add( 
    new JLabel( "RESULT ", SwingConstants.RIGHT ) );
outputField = new JTextField();
container.add( outputField );

setSize( 425, 100 );
setVisible( true );

// process GUI events
public void actionPerformed( ActionEvent event )
{
    DecimalFormat precision3 = new DecimalFormat( "0.000" );
    outputField.setText( "" );   // clear outputField
    outputField.setReadOnly( true );

    // read two numbers and calculate quotient
    try {
        number1 = Integer.parseInt( inputField1.getText() );
        number2 = Integer.parseInt( inputField2.getText() );
        result = quotient( number1, number2 );
        outputField.setText( precision3.format( result ) );
    } catch ( NumberFormatException numberFormatException ) {
        JOptionPane.showMessageDialog( this, 
            "You must enter two integers", 
            "Invalid Number Format", 
            JOptionPane.ERROR_MESSAGE );
    }

    // process improperly formatted input
    catch ( NumberFormatException numberFormatException ) {
        JOptionPane.showMessageDialog( this, 
            "You must enter two integers", 
            "Invalid Number Format", 
            JOptionPane.ERROR_MESSAGE );
    }

Fig. 14.2 A simple exception-handling example with divide by zero (part 2 of 3).
// process attempts to divide by zero

    catch ( ArithmeticException arithmeticException ) {
        JOptionPane.showMessageDialog( this,
            arithmeticException.toString(),
            "Arithmetic Exception",
            JOptionPane.ERROR_MESSAGE );
    }

    // method quotient demonstrated throwing an exception
    // when a divide-by-zero error occurs

    public double quotient( int numerator, int denominator )
    throws DivideByZeroException
    {
        if ( denominator == 0 )
            throw new DivideByZeroException();

        return ( double ) numerator / denominator;
    }

    // execute application

    public static void main( String args[] )
    {
        DivideByZeroTest application = new DivideByZeroTest();

        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }

} // end class DivideByZeroTest

----

Fig. 14.2 A simple exception-handling example with divide by zero (part 3 of 3).
When the user inputs the denominator and presses the Enter key, the program calls method `actionPerformed` (lines 54–84). Next, method `actionPerformed` proceeds with a `try` block (lines 61–67), which encloses the code that may `throw` an exception and any code that should not be executed if an exception occurs. The statements that read the integers from the `JTextField`s (lines 62–63) each use method `Integer.parseInt` to convert `Strings` to `int` values. Method `parseInt` throws a `NumberFormatException` if its `String` argument is not a valid integer. The division that can cause the divide-by-zero error is not performed explicitly in the `try` block. Rather, the call to method `quotient` (line 65) invokes the code that attempts the division. Method `quotient` (lines 89–96) `throws` the `DivideByZeroException` object, as we will see momentarily. In general, errors may surface through explicitly mentioned code in a `try` block, through calls to a method or even through deeply nested method calls initiated by code in a `try` block.

The `try` block in this example is followed by two `catch` blocks—lines 70–75 contain the exception handler for the `NumberFormatException` and lines 78–83 contain the exception handler for the `DivideByZeroException`. In general, when the program detects an exception while executing a `try` block, the program catches the exception in a `catch` block that specifies an appropriate exception type (i.e., the type in the `catch` matches the thrown exception type exactly or is a superclass of the thrown exception type). In Fig. 14.2, the first `catch` block specifies that it will catch exception objects of type `NumberFormatException` (this type matches the exception object type thrown in method `Integer.parseInt`) and the second `catch` block specifies that it will catch exception objects of type `ArithmeticException` (this type is a superclass of the exception object type thrown in method `quotient`). Only the matching `catch` handler executes when an exception occurs. Both our exception handlers simply display an error-message dialog, but exception handlers can be more elaborate than this. After executing an exception handler, program control proceeds to the first statement after the last `catch` block (or in the `finally` block, if one is present).

If the code in the `try` block does not `throw` an exception, then the `catch` handlers are skipped and execution resumes with the first line of code after the `catch` handlers (or in the `finally` block, if one is present). In Fig. 14.2, method `actionPerformed` simply returns, but the program could continue executing more statements after the `catch` blocks.

**Testing and Debugging Tip 14.5**

With exception handling, a program can continue executing after dealing with a problem. This helps ensure robust applications that contribute to what is called mission-critical computing or business-critical computing.

Now let us examine method `quotient` (lines 89–96). When the `if` structure determines that `denominator` is zero, the body of the `if` executes a `throw` statement that creates and `throws` a new `DivideByZeroException` object. This object will be caught by the `catch` block (lines 78–83) specifying type `ArithmeticException` after the `try` block. The `catch` block specifies parameter name `arithmeticException` to receive the thrown exception object. The `ArithmeticException` handler converts the exception to a `String` via `toString` and passes this `String` as the message to display in an error-message dialog.

If `denominator` is not zero, `quotient` does not throw an exception. Rather, `quotient` performs the division and returns the result of the division to the point of invocation
of method quotient in the try block (line 65). Line 66 displays the result of the calculation in the third JTextField. In this case, the try block completes successfully, so the program skips the catch blocks and the actionPerformed method completes execution normally.

Note that when quotient throws the DivideByZeroException, quotient’s block expires (i.e., the method terminates). This would cause any of its local variables to be destroyed—objects that were referenced by local variables in the block would have their reference counts decremented accordingly (and are possibly marked for garbage collection). Also, the try block from which the method was called expires before line 66 can execute. Here, too, if there were local variables created in the try block prior to the exception being thrown, these variables would be destroyed.

If a NumberFormatException is generated by lines 62–63, the try block expires and execution continues with the exception handler at line 70, which displays an error message to tell the user to input integers. Then the actionPerformed method continues with the next valid statement after the catch blocks (i.e., the method terminates in this example).

14.9 Rethrowing an Exception

It is possible that the catch handler that catches an exception may decide it cannot process the exception, or it may want to let some other catch handler handle the exception. In this case, the handler that received the exception can rethrow the exception with the statement

```java
throw exceptionReference;
```

where exceptionReference is the parameter name for the exception in the catch handler. Such a throw rethrows the exception to the next enclosing try block.

Even if a handler can process an exception, and regardless of whether it does any processing on that exception, the handler still can rethrow the exception for further processing outside the handler. A rethrown exception is detected by the next enclosing try block and is handled by an exception handler listed after that enclosing try block.

14.10 throws Clause

A throws clause lists the exceptions that can be thrown by a method as in

```java
int functionName( parameterList )
    throws ExceptionType1, ExceptionType2, ExceptionType3, ...
{
    // method body
}
```

The types of exceptions that are thrown by a method are specified in the method definition with a comma-separated list in the throws clause. A method can throw objects of the indicated classes, or it can throw objects of their subclasses.

Some exceptions can occur at any point during the execution of the program. Many of these exceptions can be avoided by coding properly. These are run-time exceptions, and they derive from class RuntimeException. For example, if your program attempts to access an out-of-range array subscript, an exception of type ArrayIndexOutOfBoundsException (derived from RuntimeException) occurs. Your program clearly can avoid such a problem; hence, it is a run-time exception.
Another run-time exception occurs when your program creates an object reference, but has not yet created an object and assigned it to the reference. Attempting to use such a null reference causes a `NullPointerException` to be thrown. Clearly, your program can avoid this circumstance; hence, it is a run-time exception. Another run-time exception is an invalid cast, which throws a `ClassCastException`.

There are a variety of exceptions that are not `RuntimeException`s. Two of the most common are `InterruptedException` (see Chapter 15, “Multithreading”) and `IOException` (see Chapter 16, “Files and Streams”).

Not all errors and exceptions that can be thrown from a method are required to be listed in the `throws` clause. `Error`s do not need to be listed, nor do `RuntimeException`s (avoidable exceptions). `Error`s are serious system problems that can occur almost anywhere, and most programs will not be able to recover from them. Methods should process `RuntimeException`s caught in their bodies directly rather than passing them on to other program components. If a method throws any non-`RuntimeException`s, it must specify those exception types in its `throws` clause.

**Software Engineering Observation 14.11**

If a non-`RuntimeException` is thrown by a method, or if that method calls methods that throw non-`RuntimeException`s, each of those exceptions must be declared in the `throws` clause of that method or caught in a `try/catch` in that method.

Java distinguishes `checked Exceptions` versus `unchecked RuntimeException`s and `Error`s. A method’s checked exceptions need to be listed in that method’s `throws` clause. `Error`s and `RuntimeException`s can be thrown from almost any method, so it would be cumbersome for programmers to be required to list them for every method definition. Such exceptions and errors are not required to be listed in a method’s `throws` clause and, hence, are said to be “unchecked” by the compiler. All non-`RuntimeException`s a method can throw must be listed in that method’s `throws` clause and, hence, are said to be “checked” by the compiler. If a non-`RuntimeException` is not listed in the throws clause, the compiler will issue an error message indicating that the exception must be caught (with a `try/catch` in the body of the method) or declared (with a `throws` clause).

**Common Programming Error 14.9**

It is a syntax error if a method throws a checked exception not in that method’s `throws` clause.

**Common Programming Error 14.10**

Attempting to throw a checked exception from a method that has no `throws` clause is a syntax error.

**Software Engineering Observation 14.12**

If your method calls other methods that explicitly `throw` checked exceptions, those exceptions must be listed in the `throws` clause of your method, unless your method catches those exceptions. This is Java’s “catch-or-declare” requirement.

**Common Programming Error 14.11**

If a subclass method overrides a superclass method, it is an error for the subclass method to list more exceptions in its `throws` list than the overridden superclass method does. A subclass’s `throws` list can contain a subset of a superclass’s `throws` list.
Java’s catch-or-declare requirement demands that the programmer either catch each checked exception or place it in the throws clause of a method. Of course, placing a checked exception in the throws clause would force other methods to process the checked exception as well. If a programmer feels a particular checked exception is unlikely to occur, the programmer might elect to catch that checked exception and do nothing with it to avoid being forced to deal with it later. This can of course come back to haunt you, because as a program evolves, it may become important to deal with this checked exception.

**Testing and Debugging Tip 14.6**

Do not try to circumvent Java’s catch-or-declare requirement by simply catching exceptions and doing nothing with them. Exceptions are generally of a serious enough nature that they need to be processed rather than suppressed.

**Testing and Debugging Tip 14.7**

The Java compiler, through the throws clause used with exception handling, forces programmers to process the exceptions that can be thrown from each method a program calls. This helps avoid bugs that arise in programs when programmers ignore the fact that problems occur and make no provisions for these problems.

**Software Engineering Observation 14.13**

Subclass methods that do not override their corresponding superclass methods exhibit the same exception-handling behavior of the inherited superclass methods. The throws clause of a subclass method that overrides a superclass method may contain the same list of exceptions as the overridden superclass method or a subset of that list.

**Common Programming Error 14.12**

The Java compiler requires that a method either catch any checked exceptions thrown in the method (either directly from the method’s body or indirectly through called methods) or declare checked Exception: the method can throw to other methods; otherwise, the Java compiler issues a syntax error.

**Testing and Debugging Tip 14.8**

Suppose a method throws all subclasses of a particular exception superclass. You may be tempted to list only the superclass in the throws clause. Instead, explicitly list all the subclasses. This focuses the programmer’s attention on the specific Exception: that may occur and will often help avoid bugs caused by performing general processing for a category of exception types.

Figure 14.3–Fig. 14.8 list many of Java’s Errors and Exceptions hierarchically for the packages java.lang, java.util, java.io, java.awt and java.net. In these tables, a class indented under another class is a subclass. The exception and error classes for the other packages of the Java API can be found in the Java online documentation. The online documentation for each method in the API specifies whether that method throws exceptions and what the exceptions are that can be thrown. We show a portion of Java’s Error hierarchy in Fig. 14.3. Most Java programmers ignore Errors. They are serious but rare events.

Figure 14.4 is particularly important because it lists many of Java’s RuntimeExceptions. Although Java programmers are not required to declare these exceptions in throws clauses, these are the exceptions that commonly will be caught and handled in Java applications.
The java.lang package errors

**Error** (all in java.lang except for AWTErrort, which is in java.awt)

- LinkageError
  - ClassCircularityError
  - ClassFormatError
  - ExceptionInInitializerError
  - IncompatibleClassChangeError
    - AbstractMethodError
    - IllegalAccessError
    - InstantiationException
    - NoSuchFieldError
    - NoSuchMethodError
  - NoClassDefFoundError
  - UnsatisfiedLinkError
  - VerifyError
- ThreadDeath
- VirtualMachineError (Abstract class)
  - InternalError
  - OutOfMemoryError
  - StackOverflowError
  - UnknownError
- AWTErrort (package java.awt)

---

Fig. 14.3 The java.lang package errors.

The java.lang package exceptions

**Exception**

- ClassNotFoundException
- CloneNotSupportedException
- IllegalAccessException
- InstantiationException
- InterruptedException
- NoSuchFieldException
- NoSuchMethodException
- NOpaqueException
- NoSuchMethodError

---

Fig. 14.4 The java.lang package exceptions (part 1 of 2).
Figure 14.5 lists Java’s other three **RuntimeExceptions** data types. We will encounter these exceptions in Chapter 20 when we study the **Vector** class. A **Vector** is a dynamic array that can grow and shrink to accommodate a program’s varying storage requirements.

Figure 14.6 lists Java’s **IOExceptions**. These are all checked exceptions that can occur during input/output and file processing.

Figure 14.7 lists the **java.awt** package’s only checked **Exception**, the **AWTException**. This is a checked exception that is thrown by various abstract windowing toolkit methods.

---

**The java.lang package exceptions**

- `RuntimeException`
  - `ArithmeticException`
  - `ArrayStoreException`
  - `ClassCastException`
  - `IllegalArgumentException`
  - `IllegalMonitorStateException`
  - `IllegalStateException`
  - `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
  - `NegativeArraySizeException`
  - `NullPointerException`
  - `SecurityException`

---

**Fig. 14.4** The **java.lang** package exceptions (part 2 of 2).

---

**The java.util package exceptions**

- `Exception`
  - `RuntimeException`
  - `EmptyStackException`
  - `MissingResourceException`
  - `NoSuchElementException`
  - `TooManyListenersException`

---

**Fig. 14.5** The **java.util** package exceptions.
The **java.io** package exceptions

<table>
<thead>
<tr>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOException</td>
</tr>
<tr>
<td>CharConversionException</td>
</tr>
<tr>
<td>EOFException</td>
</tr>
<tr>
<td>FileNotFoundException</td>
</tr>
<tr>
<td>InterruptedIOException</td>
</tr>
<tr>
<td>ObjectStreamException</td>
</tr>
<tr>
<td>InvalidClassException</td>
</tr>
<tr>
<td>InvalidObjectException</td>
</tr>
<tr>
<td>NotActiveException</td>
</tr>
<tr>
<td>NotSerializableException</td>
</tr>
<tr>
<td>OptionalDataException</td>
</tr>
<tr>
<td>StreamCorruptedException</td>
</tr>
<tr>
<td>WriteAbortedException</td>
</tr>
<tr>
<td>SyncFailedException</td>
</tr>
<tr>
<td>UnsupportedEncodingException</td>
</tr>
<tr>
<td>UTFDataFormatException</td>
</tr>
</tbody>
</table>

**Fig. 14.6** The **java.io** package exceptions.

The **java.awt** package exceptions

<table>
<thead>
<tr>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWTException</td>
</tr>
<tr>
<td>RuntimeException</td>
</tr>
<tr>
<td>IllegalStateException</td>
</tr>
<tr>
<td>IllegalComponentStateException</td>
</tr>
</tbody>
</table>

**Fig. 14.7** The **java.awt** package exceptions.

Figure 14.8 lists the **IOExceptions** of the **java.net** package. These are all checked **Exceptions** that indicate various networking problems.

Most packages in the Java API define **Exceptions** and **Errors** specific to the package. For a complete list of these types, see the online API documentation for the package.
14.11 Constructors, Finalizers and Exception Handling

First, let us deal with an issue we have mentioned, but that has yet to be resolved satisfactorily. What happens when an error is detected in a constructor? The problem is that a constructor cannot return a value, so how do we let the program know that an object has not been constructed properly? One scheme is to return the improperly constructed object and hope that anyone using the object would perform appropriate tests to determine that the object is in fact bad. However, this directly contradicts discussions in Chapter 8 in which we indicated that you should maintain an object in a consistent state at all times. Another scheme is to set some instance variable with an error indicator outside the constructor, but this is a poor programming practice. In Java, the typical (and proper) mechanism is to throw an exception from the constructor to the code creating the object. The thrown object contains the information about the failed constructor call and the caller is responsible for handling the failure.

When an exception occurs in a constructor, other objects created by that constructor are marked for eventual garbage collection. Before each object is garbage collected, its finalize method will be called.

14.12 Exceptions and Inheritance

Various exception classes can be derived from a common superclass. If a catch is written to catch exception objects of a superclass type, it can also catch all objects of subclasses of that superclass. This can allow for polymorphic processing of related exceptions.

Using inheritance with exceptions enables an exception handler to catch related errors with a concise notation. One could certainly catch each subclass exception object individually if those exceptions require different processing, but it is more concise to catch the superclass exception object. Of course, this makes sense only if the handling behavior would be the same for all subclasses. Otherwise, catch each subclass exception individually.
Testing and Debugging Tip 14.9

Catching subclass exception objects individually is subject to error if the programmer forgets to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Often, a catch of the superclass type follows all other subclass exception handlers to ensure that all exceptions are processed properly.

14.13 finally Block

Programs that obtain certain types of resources must return those resources to the system explicitly to avoid so-called resource leaks. In programming languages like C and C++, the most common kind of resource leak is a memory leak. Java performs automatic garbage collection of memory no longer used by programs, thus avoiding most memory leaks. However, other types of resource leaks can occur in Java.

Software Engineering Observation 14.14

A finally block typically contains code to release resources acquired in its corresponding try block; this is an effective way to eliminate resource leaks. For example, the finally block should close any files opened in the try block.

Testing and Debugging Tip 14.10

Actually, Java does not eliminate memory leaks completely. There is a subtle issue here. Java will not garbage collect an object until there are no more references to the object. Thus, memory leaks can occur, but only if programmers erroneously keep references to unwanted objects. Most memory leak problems are solved by Java’s garbage collection.

The finally block is optional. If it is present, it is placed after the last of a try block’s catch blocks, as follows:

```java
try {
    statements
    resource-acquisition statements
} catch (AKindOfException exception1) {
    exception-handling statements
} catch (AnotherKindOfException exception2) {
    exception-handling statements
} finally {
    statements
    resource-release statements
}
```

Java guarantees that a finally block (if one is present) will execute regardless of whether any exception is thrown in the corresponding try block or any of its corresponding catch blocks. Java also guarantees that a finally block (if one is present) will execute if a try block exits via a return, break or continue statement.

Resource-release code is placed in a finally block. Suppose a resource is allocated in a try block. If no exception occurs, the catch handlers are skipped and control proceeds to the finally block, which frees the resource. Control then proceeds to the first statement after the finally block.
If an exception occurs, the program skips the rest of the `try` block. If the program catches the exception in one of the `catch` handlers, the program processes the exception. Then the `finally` block releases the resource, and control then proceeds to the first statement after the `finally` block.

If an exception that occurs in the `try` block cannot be caught by one of the `catch` handlers, the program skips the rest of the `try` block and control proceeds to the `finally` block, which releases the resource. Then the program passes the exception up the call chain until some calling method chooses to `catch` it. If no method chooses to deal with it, a non-GUI-based application terminates.

If a `catch` handler throws an exception, the `finally` block still executes. Then the exception is passed up the call chain for a calling method to `catch` and handle.

The Java application of Fig. 14.9 demonstrates that the `finally` block (if one is present) executes even if an exception is not thrown in the corresponding `try` block. The program contains methods `main` (lines 7–21), `throwException` (lines 24–50) and `doesNotThrowException` (lines 53–75). Methods `throwException` and `doesNotThrowException` are declared `static` so `main` (another `static` method) can call them directly.

Method `main` begins executing, enters its `try` block and immediately calls `throwException` (line 11). Method `throwException` throws an `Exception` (line 29), catches it (line 33) and rethrows it (line 37). The rethrown exception will be handled in `main`, but first the `finally` block (lines 44–47) executes. Method `main` detects the rethrown exception in the `try` block in `main` (lines 10–12) and handles it by the `catch` block (lines 15–18). Next, `main` calls method `doesNotThrowException` (line 20). No exception is thrown in `doesNotThrowException`’s `try` block, so the program skips the `catch` block (lines 61–64), but the `finally` block (lines 68–61) nevertheless executes. Control proceeds to the statement after the `finally` block. Then control returns to `main` and the program terminates.

```
// Fig. 14.9: UsingExceptions.java
// Demonstration of the try-catch-finally
// exception-handling mechanism.
public class UsingExceptions {

    // execute application
    public static void main(String args[]) {
        // call method throwException
        try {
            throwException();
        }
        // catch Exceptions thrown by method throwException
        catch (Exception exception) {
            System.err.println( "Exception handled in main" );
        }

    }

Fig. 14.9 Demonstration of the try-catch-finally exception-handling mechanism (part 1 of 3).
```
```java
public static void throwException() throws Exception
{
    // throw an exception and immediately catch it
    try {
        System.out.println("Method throwException");
        throw new Exception(); // generate exception
    }

    // catch exception thrown in try block
    catch (Exception exception)
    {
        System.err.println("Exception handled in method throwException");
        throw exception; // rethrow for further processing
    }

    // any code here would not be reached
}

// this block executes regardless of what occurs in // try/catch
finally {
    System.err.println("Finally executed in throwException");
}

// any code here would not be reached
}

// demonstrate finally when no exception occurs
public static void doesNotThrowException()
{
    // try block does not throw an exception
    try {
        System.out.println("Method doesNotThrowException");
    }

    // catch does not execute, because no exception thrown
    catch( Exception exception )
    {
        System.err.println( exception.toString() );
    }

    // this block executes regardless of what occurs in // try/catch
    finally {
        System.err.println("Finally executed in doesNotThrowException");
    }
}
```

Fig. 14.9 Demonstration of the **try-catch-finally** exception-handling mechanism (part 2 of 3).
The Java application in Fig. 14.10 demonstrates that when an exception thrown in a `try` block is not caught in a corresponding `catch` block, the exception will be detected in the next outer `try` block and handled by an appropriate `catch` block (if one is present) associated with that outer `try` block.

```java
public class UsingExceptions {

    public static void main( String args[] ) {

        // call throwException to demonstrate stack unwinding
        try {
            throwException();
        }

        // catch exception thrown in throwException
        catch ( Exception exception ) {
            System.err.println( "Exception handled in main" );
        }

        // throwException throws an exception that is not caught in
        // the body of this method
        public static void throwException() throws Exception {

            // throw an exception and catch it in main
            try {
                System.out.println( "Method throwException" );
                throw new Exception();      // generate exception
            }

        }

    }

}
```

Fig. 14.10 Demonstration of stack unwinding (part 1 of 2).
When method `main` executes, line 10 in the `try` block calls `throwException` (lines 21–39). In the `try` block of method `throwException`, line 26 throws an `Exception`. This terminates the `try` block immediately and control proceeds to the `catch` handler at line 30. The type being caught (`RuntimeException`) is not an exact match with the thrown type (`Exception`) and is not a superclass of the thrown type, so the exception is not caught in method `throwException`. The exception must be handled before normal program execution can continue. Therefore, method `throwException` terminates (but not until its `finally` block executes) and returns control to the point from which it was called in the program (line 10). Line 10 is in the enclosing `try` block. If the exception has not yet been handled, the `try` block terminates and an attempt is made to catch the exception at line 14. The type being caught (`Exception`) matches the thrown type. Therefore, the `catch` handler processes the exception and the program terminates at the end of `main`.

As we have seen, a `finally` block executes for a variety of reasons, such as a `try` completing successfully, handling of an exception in a local `catch`, an exception being thrown for which no local `catch` is available or execution of a program control statement like a `return`, `break` or `continue`. Normally, the `finally` block executes, then behaves appropriately (we will call this `finally`'s “continuation action”) depending on the reason program control entered the block. For example, if an exception is thrown in the `finally` block, the continuation action will be for that exception to be processed in the next enclosing `try` block. Unfortunately, if there was an exception that had not yet been caught, that exception is lost and the more recent exception is processed. This is dangerous.

**Common Programming Error 14.13**

If an exception is thrown for which no local `catch` is available, when control enters the local `finally` block, the `finally` block could also throw an exception. If this happens, the first exception will be lost.
Exception Handling Chapter 14

Testing and Debugging Tip 14.11
Avoid placing code that can throw an exception in a finally block. If such code is required, enclose the code in a try/catch within the finally block.

Good Programming Practice 14.7
Java’s exception-handling mechanism is intended to remove error-processing code from the main line of a program’s code to improve program clarity. Do not place try/catch/finally around every statement that may throw an exception. This makes programs difficult to read. Rather, place one try block around a significant portion of your code, follow that try block with catch blocks that handle each possible exception and follow the catch blocks with a single finally block (if one is required).

Performance Tip 14.3
As a rule, resources should be released as soon as it is apparent that they are no longer needed. This makes these resources immediately available for reuse and can improve program performance.

Software Engineering Observation 14.15
If a try block has a corresponding finally block, the finally block will execute even if the try block exits with return, break or continue; then the effect of the return, break or continue will occur.

14.14 Using printStackTrace and getMessage
Exceptions derive from class Throwable. Class Throwable offers a printStackTrace method that prints the method call stack. By calling this method for a Throwable object that has been caught, a program can print the method call stack. Often, this is helpful in testing and debugging. Two other overloaded versions of printStackTrace enable the program to direct the stack trace to a PrintStream or PrintWriter stream. In this section, we consider an example that exercises the printStackTrace method and another useful method, getMessage.

Testing and Debugging Tip 14.12
All Throwable objects contain a printStackTrace method that prints a stack trace for the object.

Testing and Debugging Tip 14.13
An exception that is not caught eventually causes Java’s default exception handler to run. This displays the name of the exception, the optional character string that was supplied when the exception was constructed and a complete execution stack trace. The stack trace shows the complete method call stack. This lets the programmer see the path of execution that led to the exception file-by-file (and thus class-by-class) and method-by-method. This information is helpful in debugging a program.

There are two constructors for class Throwable. The first constructor

```java
public Throwable()
```

takes no arguments. The second constructor

```java
public Throwable( String informationString )
```
Chapter 14 Exception Handling

takes the argument `informationString`, which is descriptive information about the `Throwable` object. The `informationString` stored in the `Throwable` object may be obtained with method `getMessage`.

Testing and Debugging Tip 14.14

`Throwable` classes have a constructor that accepts a `String` argument. Using this form of the constructor is helpful in determining the source of the exception via method `getMessage`.

Figure 14.11 demonstrates `getMessage` and `printStackTrace`. Method `getMessage` returns the descriptive `String` stored in an exception. Method `printStackTrace` outputs to the standard error stream (normally, the command line or console) an error message with the class name of the exception, the descriptive `String` stored in the exception and a list of the methods that had not completed execution when the exception was thrown (i.e., all methods currently residing on the method call stack).

In the program, `main` invokes `method1`, `method1` invokes `method2` and `method2` invokes `method3`. At this point, the method call stack for the program is

```
method3
method2
method1
main
```

with the last method called (`method3`) at the top and the first method called (`main`) at the bottom. When `method3` throws an `Exception` (line 36), a stack-trace message is generated and stored in the `Exception` object. The stack trace reflects the throw point in the code (i.e., line 36). Then, the stack unwinds to the first method in the method call stack in which the exception can be caught (i.e., `main` because it contains a `catch` handler for `Exception`). The `catch` handler then uses `getMessage` and `printStackTrace` on the `Exception` object `exception` to produce the output. Notice that the line numbers in the output window correspond to the line numbers in the program.

```
1 // Fig. 14.11: UsingExceptions.java
2 // Demonstrating the getMessage and printStackTrace
3 // methods inherited into all exception classes.
4 public class UsingExceptions {
5
6     // execute application
7     public static void main( String args[] )
8     {
9         // call method1
10         try {
11             method1();
12         }
13     }
14     // catch Exceptions thrown from method1
15     catch ( Exception exception ) {
16         System.err.println( exception.getMessage() + "\n" );
17         exception.printStackTrace();
18     }
19 }
```

Fig. 14.11 Using `getMessage` and `printStackTrace` (part 1 of 2).
**SUMMARY**

- Some common examples of exceptions are memory exhaustion, an out-of-bounds array subscript, arithmetic overflow, division by zero and invalid method parameters.

- Exception handling is designed for dealing with synchronous malfunctions (i.e., those that occur as the result of a program’s execution).

- Exception handling is used in situations in which a malfunction will be dealt with in a different scope from that which detected the malfunction.

- Exception handling should be used to process exceptions from software components such as methods, libraries and classes that are likely to be widely used and where it does not make sense for those components to handle their own exceptions.

- Exception handling should be used on large projects to handle error processing in a standardized manner for the entire project.

- Java exception handling is geared to situations in which the method that detects an error is unable to deal with it. Such a method will **throw** an exception. If the exception matches the type of the parameter in one of the **catch** blocks, the code for that **catch** block executes.

- The programmer encloses in a **try** block the code that may generate an exception. The **try** block is followed by one or more **catch** blocks. Each **catch** block specifies the type of exception it can catch and handle. Each **catch** block is an exception handler.
Chapter 14 Exception Handling

- If no exceptions are thrown in the `try` block, the exception handlers for that block are skipped. Then the program resumes execution after the last `catch` block, after executing a `finally` block if one is provided.

- Exceptions are thrown in a `try` block in a method or from a method called directly or indirectly from the `try` block.

- The operand of a `throw` can be of any class derived from `Throwable`. The immediate subclasses of `Throwable` are `Error` and `Exception`.

- `RuntimeException`s and `Error`s are said to be “unchecked.” Non-`RuntimeException`s are said to be “checked.” The checked exceptions thrown by a particular method must be specified in that method’s `throws` clause.

- Exceptions are caught by the closest exception handler (for the `try` block from which the exception was thrown) specifying an appropriate type.

- An exception terminates the block in which the exception occurred.

- A handler may rethrow the object to an outer `try` block.

- `catch(Exception exception)` catches all `Exceptions`.

- `catch(Throwable throwable)` catches all `Exceptions` and `Errors`.

- If no handler matches a particular thrown object, the search for a match continues in an enclosing `try` block.

- Exception handlers are searched in order for an appropriate match based on type. The first handler that matches is executed. When that handler finishes executing, control resumes with the first statement after the last `catch` block.

- The order of the exception handlers affects how an exception is handled.

- A subclass object can be caught either by a handler specifying that subclass type or by handlers specifying the types of any direct or indirect superclasses of that subclass.

- If no handler is found for an exception, a non-GUI-based application terminates; an applet or a GUI-based application will return to its regular event handling.

- An exception handler cannot access variables in the scope of its `try` block because by the time the exception handler begins executing, the `try` block has expired. Information the handler needs is normally passed in the thrown object.

- Exception handlers can rethrow an exception. They can convert one type of exception into another by throwing a different exception. They can perform any necessary recovery and resume execution after the last exception handler. They can look at the situation causing the error, remove the cause of the error and retry by calling the original method that caused an exception. They can simply return some status value to their environment.

- A handler that catches a subclass object should be placed before a handler that catches a superclass object. If the superclass handler were first, it would catch superclass objects and the objects of subclasses of that superclass.

- When an exception is caught, it is possible that resources may have been allocated, but not yet released in the `try` block. A `finally` block should release these resources.

- It is possible that the handler that catches an exception may decide it cannot process the exception. In this case, the handler can simply rethrow the exception. A `throw` followed by the exception object name rethrows the exception.

- Even if a handler can process an exception, and regardless of whether it does any processing on that exception, the handler can rethrow the exception for further processing outside the handler. A rethrown exception is detected by the next enclosing `try` block (normally in a calling method) and is
handled by an appropriate exception handler (if there is one) listed after that enclosing try block.

- A throws clause lists the checked exceptions that may be thrown from a method. A method may throw the indicated exceptions, or it may throw subclass types. If a checked exception not listed in the throws clause is thrown, a syntax error occurs.

- A powerful reason for using inheritance with exceptions is to catch a variety of related errors easily with concise notation. One could certainly catch each type of subclass exception object individually, but it is more concise to simply catch the superclass exception object.

**TERMINOLOGY**

- ArithmeticException
- ArrayIndexOutOfBoundsException
- Array exceptions
- catch a group of exceptions
- catch all exceptions
- catch an exception
- catch block
- catch(Exception e)
- catch-or-declare requirement
- checked Exceptions
- ClassCastException
- declare exceptions that can be thrown default exception handler
- EmptyStackException
- Error class
- Error class hierarchy
- error handling exception
- Exception class
- Exception class hierarchy
- exception handler
- exception handling exception
- exception object
- fault tolerance
- FileNotFoundException
- finally block
- getMessage method of Throwable class
- handle an exception
- IncompatibleAccessException
- IncompatibleClassChangeException
- instanceof operator
- InstantiationException
- InternalException
- InterruptedException
- IOException
- library exception classes
- memory exhaustion
- mission-critical computing
- NegativeArraySizeException
- NoClassDefFoundException
- non-run-time exception
- NullPointerException
- OutOfMemoryError
- printStackTrace method (Throwable)
- rethrow an exception
- RuntimeException
- stack unwinding
- synchronous error
- termination model of exception handling
- throw an exception
- throw point
- throw statement
- Throwable class
- throws clause
- try block
- unchecked Exceptions
- UnsatisfiedLinkException

**SELF-REVIEW EXERCISES**

14.1 List five common examples of exceptions.
14.2 Why should exception-handling techniques not be used for conventional program control?
14.3 Why are exceptions particularly appropriate for dealing with errors produced by library classes and methods?
14.4 What is a “resource leak?”
Chapter 14

14.5 If no exceptions are thrown in a try block, where does control proceed to when the try block completes execution?

14.6 What happens if an exception occurs and an appropriate exception handler cannot be found?

14.7 Give a key advantage of using catch(Exception e).

14.8 Should a conventional applet or application catch Error objects?

14.9 What happens if several handlers match the type of the thrown object?

14.10 Why would a programmer specify a superclass type as the type of a catch handler, then throw objects of subclass types?

14.11 How might a catch handler be written to process related types of errors without using inheritance among exception classes?

14.12 What is the key reason for using finally blocks?

14.13 Does throwing an Exception have to cause program termination?

14.14 What happens when a catch handler throws an Exception?

14.15 What happens to a local reference in a try block when that block throws an Exception?

ANSWERS TO SELF-REVIEW EXERCISES

14.1 Memory exhaustion, array subscript out of bounds, arithmetic overflow, division by zero, invalid method parameters.

14.2 (a) Exception handling is designed to handle infrequently occurring situations that often result in program termination, so compiler writers are not required to implement exception handling to perform optimally. (b) Flow of control with conventional control structures is generally clearer and more efficient than with exceptions. (c) Problems can occur because the stack is unwound when an exception occurs and resources allocated prior to the exception may not be freed. (d) The “additional” exceptions can get in the way of genuine error-type exceptions. It becomes more difficult for the programmer to keep track of the larger number of exception cases.

14.3 It is unlikely that library classes and methods could perform error processing that would meet the unique needs of all users.

14.4 A resource leak occurs when an executing program does not properly release a resource when the resource is no longer needed. If the program attempts to use the resource again in the future, the program may not be able to access the resource.

14.5 The exception handlers (in the catch blocks) for that try block are skipped, and the program resumes execution after the last catch block. If there is a finally block, it is executed and the program resumes execution after the finally block.

14.6 A non-GUI-based application terminates; an applet or a GUI-based application resumes regular event processing.

14.7 The form catch(Exception e) catches any type of exception thrown in a try block. An advantage is that no thrown Exception can slip by.

14.8 Errors are usually serious problems with the underlying Java system; most programs will not want to catch Errors.

14.9 The first matching Exception handler after the try block is executed.

14.10 This is a nice way to catch related types of exceptions, but it should be used carefully.
14.11 Provide a single Exception subclass and catch handler for a group of exceptions. As each exception occurs, the exception object can be created with different instance data. The catch handler can examine this data to distinguish the type of the Exception.

14.12 The finally block is the preferred means for preventing resource leaks.

14.13 No, but it does terminate the block in which the Exception is thrown.

14.14 The exception will be processed by a catch handler (if one exists) associated with the try block (if one exists) enclosing the catch handler that caused the exception.

14.15 The reference is removed from memory, and the reference count for the referenced object is decremented. If the reference count is zero, the object is marked for garbage collection.

**EXERCISES**

14.16 Under what circumstances would you use the following catch handler?

```java
    catch ( Exception exception ) {
        throw exception;
    }
```

14.17 List the benefits of exception handling over conventional means of error processing.

14.18 Describe an object-oriented technique for handling related exceptions.

14.19 Until this chapter, we have found that dealing with errors detected by constructors is a bit awkward. Explain why exception handling is an effective means for dealing with constructor failure.

14.20 Suppose a program throws an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself throws the same exception. Does this create an infinite recursion? Explain your answer.

14.21 Use inheritance to create an exception superclass and various exception subclasses. Write a program to demonstrate that the catch specifying the superclass catches subclass exceptions.

14.22 Write a Java program that shows that not all finalizers for objects constructed in a block are necessarily called after an exception is thrown from that block.

14.23 Write a Java program that demonstrates how various exceptions are caught with

```java
    catch ( Exception exception )
```

14.24 Write a Java program that shows that the order of exception handlers is important. If you try to catch a superclass exception type before a subclass type, the compiler should generate errors. Explain why these errors occur.

14.25 Write a Java program that shows a constructor passing information about constructor failure to an exception handler after a try block.

14.26 Write a Java program that illustrates rethrowing an exception.

14.27 Write a Java program that shows that a method with its own try block does not have to catch every possible error generated within the try. Some exceptions can slip through to, and be handled in, other scopes.
Objectives

• To understand the notion of multithreading.
• To appreciate how multithreading can improve performance.
• To understand how to create, manage and destroy threads.
• To understand the life cycle of a thread.
• To study several examples of thread synchronization.
• To understand thread priorities and scheduling.
• To understand daemon threads and thread groups.

The spider’s touch, how exquisitely fine!
Feels at each thread, and lives along the line.
Alexander Pope

A person with one watch knows what time it is; a person with two watches is never sure.
Proverb

Conversation is but carving!
Give no more to every guest,
Than he’s able to digest.
Jonathan Swift

Learn to labor and to wait.
Henry Wadsworth Longfellow

The most general definition of beauty…Multeity in Unity.
Samuel Taylor Coleridge
15.1 Introduction

It would be nice if we could “do one thing at a time” and “do it well,” but that is simply not how the world works. The human body performs a great variety of operations in parallel, or as we will say throughout this chapter, concurrently. Respiration, blood circulation and digestion, for example, can occur concurrently. All of the senses—seeing, touching, smelling, tasting and hearing—can all occur concurrently. An automobile can be accelerating, turning, air conditioning and playing music concurrently. Computers, too, perform operations concurrently. It is common today for desktop personal computers to be compiling a program, printing a file and receiving e-mail messages over a network concurrently.

Concurrency is important in our lives. Ironically, though, most programming languages do not enable programmers to specify concurrent activities. Rather, programming languages generally provide only a simple set of control structures that enable programmers to perform one action at a time then proceed to the next action after the previous one is finished. The kind of concurrency that computers perform today normally is implemented as operating systems “primitives” available only to highly experienced “systems programmers.”

The Ada programming language developed by the United States Department of Defense made concurrency primitives widely available to defense contractors building command and control systems. But Ada has not been widely used in universities and commercial industry.

Java is unique among popular general-purpose programming languages in that it makes concurrency primitives available to the applications programmer. The programmer specifies that applications contain threads of execution, each thread designating a portion of a program that may execute concurrently with other threads. This capability, called multithreading, gives the Java programmer powerful capabilities not available in C and C++, the languages on which Java is based. C and C++ are called single-threaded languages.
Software Engineering Observation 15.1

Unlike many languages that do not have built-in multithreading (such as C and C++) and must therefore make calls to operating system multithreading primitives, Java includes multithreading primitives as part of the language itself (actually in classes `Thread`, `ThreadGroup`, `ThreadLocal` and `ThreadDeath` of the `java.lang` package). This encourages the use of multithreading among a larger part of the applications-programming community.

We will discuss many applications of concurrent programming. When programs download large files such as audio clips or video clips from the World Wide Web, we do not want to wait until an entire clip is downloaded before starting the playback. So we can put multiple threads to work: One that downloads a clip and another that plays the clip so that these activities, or tasks, may proceed concurrently. To avoid choppy playback, we will coordinate the threads so that the player thread does not begin until there is a sufficient amount of the clip in memory to keep the player thread busy.

Another example of multithreading is Java’s automatic garbage collection. In C and C++, the programmer is responsible for reclaiming dynamically allocated memory. Java provides a garbage collector thread that reclaims dynamically allocated memory that the program no longer needs.

Testing and Debugging Tip 15.1

In C and C++, programmers must provide explicit statements that reclaim dynamically allocated memory. When memory is not reclaimed (because a programmer forgets to do so, because of a logic error or because an exception diverts program control), this results in an all-too-common error called a memory leak that can eventually exhaust the supply of free memory and may cause premature program termination. Java’s automatic garbage collection eliminates the vast majority of memory leaks, that is, those that are due to orphaned (un-referenced) objects.

Java’s garbage collector runs as a low-priority thread. When Java determines that there are no longer any references to an object, it marks the object for eventual garbage collection. The garbage-collector thread runs when processor time is available and when there are no higher priority runnable threads. However, the garbage collector will run immediately when the system is out of memory.

Performance Tip 15.1

Setting an object reference to `null` marks that object for eventual garbage collection (if there are no other references to the object). This can help conserve memory in a system in which a local variable that refers to an object does not go out of scope because the method in which it appears executes for a lengthy period.

Writing multithreaded programs can be tricky. Although the human mind can perform many functions concurrently, humans find it difficult to jump between parallel “trains of thought.” To see why multithreading can be difficult to program and understand, try the following experiment: open three books to page 1. Now try reading the books concurrently. Read a few words from the first book, then read a few words from the second book, then read a few words from the third book, then loop back and read the next few words from the first book, and so on. After a brief time, you will appreciate the challenges of multithreading: switching between books, reading briefly, remembering your place in each
Performance Tip 15.2

A problem with single-threaded applications is that lengthy activities must complete before other activities can begin. In a multithreaded application, threads can share a processor (or set of processors), so that multiple tasks are performed in parallel.

Although Java is perhaps the world’s most portable programming language, certain portions of the language are nevertheless platform dependent. In particular, there are differences among the first three Java platforms implemented, namely the Solaris implementation and the Win32 implementations (i.e., Windows-based implementations for Windows 95 and Windows NT).

The Solaris Java platform runs a thread of a given priority to completion or until a higher priority thread becomes ready. At that point preemption occurs (i.e., the processor is given to the higher priority thread while the previously running thread must wait).

In the 32-bit Java implementations for Windows 95 and Windows NT, threads are timesliced. This means that each thread is given a limited amount of time (called a time quantum) to execute on a processor, and when that time expires the thread is made to wait while all other threads of equal priority get their chances to use their quantum in round-robin fashion. Then the original thread resumes execution. Thus, on Windows 95 and Windows NT, a running thread can be preempted by a thread of equal priority; whereas, on the Solaris implementation, a running Java thread can only be preempted by a higher priority thread. Future Solaris Java systems are expected to perform timeslicing as well.

Portability Tip 15.1

Java multithreading is platform dependent. Thus, a multithreaded application could behave differently on different Java implementations.

15.2 Class Thread: An Overview of the Thread Methods

In this section, we overview the various thread-related methods in the Java API. We use many of these methods in live-code examples throughout the chapter. The reader should refer to the Java API directly for more details on using each method, especially the exceptions thrown by each method.

Class Thread (package java.lang) has several constructors. The constructor

```java
public Thread( String threadName )
```

constructs a Thread object whose name is threadName. The constructor

```java
public Thread()
```

constructs a Thread whose name is "Thread-" concatenated with a number, like Thread-1, Thread-2, and so on.

The code that “does the real work” of a thread is placed in its run method. The run method can be overridden in a subclass of Thread or it may be implemented in a Runnable object; Runnable is an important Java interface that we study in Section 15.10.

A program launches a thread’s execution by calling the thread’s start method, which, in turn, calls method run. After start launches the thread, start returns to its
caller immediately. The caller then executes concurrently with the launched thread. The `start` method throws an `IllegalThreadStateException` if the thread it is trying to start has already been started.

The `static` method `sleep` is called with an argument specifying how long the currently executing thread should sleep (in milliseconds); while a thread sleeps, it does not contend for the processor, so other threads can execute. This can give lower priority threads a chance to run.

The `interrupt` method is called to interrupt a thread. The `static` method `interrupted` returns `true` if the current thread has been interrupted and `false` otherwise. A program can invoke a specific thread’s `isInterrupted` method to determine whether that thread has been interrupted.

Method `isAlive` returns `true` if `start` has been called for a given thread and the thread is not dead (i.e., its controlling `run` method has not completed execution).

Method `setName` sets a `Thread`’s name. Method `getName` returns the name of the `Thread`. Method `toString` returns a `String` consisting of the name of the thread, the priority of the thread and the thread’s `ThreadGroup` (discussed in Section 15.11).

The `static` method `currentThread` returns a reference to the currently executing `Thread`.

Method `join` waits for the `Thread` to which the message is sent to die before the calling `Thread` can proceed; no argument or an argument of 0 milliseconds to method `join` indicates that the current `Thread` will wait forever for the target `Thread` to die before the calling `Thread` proceeds. Such waiting can be dangerous; it can lead to two particularly serious problems called deadlock and indefinite postponement. We will discuss these momentarily.

**Testing and Debugging Tip 15.2**

Method `dumpStack` is useful for debugging multithreaded applications. A program calls `static` method `dumpStack` to print a method-call stack trace for the current `Thread`.

### 15.3 Thread States: Life Cycle of a Thread

At any time, a thread is said to be in one of several thread states (illustrated in Fig. 15.1). Let us say that a thread that was just created is in the born state. The thread remains in this state until the program calls the thread’s `start` method, which causes the thread to enter the ready state (also known as the runnable state). The highest priority ready thread enters the running state (i.e., the thread begins executing), when the system assigns a processor to the thread. A thread enters the dead state when its `run` method completes or terminates for any reason—a dead thread eventually will be disposed of by the system.

One common way for a running thread to enter the blocked state is when the thread issues an input/output request. In this case, a blocked thread becomes ready when the I/O for which it is waiting completes. A blocked thread cannot use a processor even if one is available.

When the program calls method `sleep` in a running thread, that thread enters the sleeping state. A sleeping thread becomes ready after the designated sleep time expires. A sleeping thread cannot use a processor even if one is available. If the program calls method `interrupt` on a sleeping thread, that thread exits the sleeping state and becomes ready to execute.
When a running thread calls `wait`, the thread enters a waiting state for the particular object on which `wait` was called. One thread in the waiting state for a particular object becomes ready on a call to `notify` issued by another thread associated with that object. Every thread in the waiting state for a given object becomes ready on a call to `notifyAll` by another thread associated with that object. The `wait`, `notify` and `notifyAll` methods will be discussed in more depth shortly, when we consider monitors.

A thread enters the `dead state` when its `run` method either completes or throws an uncaught exception.

### 15.4 Thread Priorities and Thread Scheduling

Every Java applet or application is multithreaded. Every Java thread has a priority in the range `Thread.MIN_PRIORITY` (a constant of 1) and `Thread.MAX_PRIORITY` (a constant of 10). By default, each thread is given priority `Thread.NORM_PRIORITY` (a constant of 5). Each new thread inherits the priority of the thread that creates it.

Some Java platforms support a concept called `timeslicing` and some do not. Without timeslicing, each thread in a set of equal-priority threads runs to completion (unless the thread leaves the running state and enters the waiting, sleeping or blocked state, or the
thread gets interrupted by a higher priority thread) before that thread’s peers get a chance to execute. With timeslicing, each thread receives a brief burst of processor time called a \textit{quantum} during which that thread can execute. At the completion of the quantum, even if that thread has not finished executing, the operating system takes the processor away from that thread and gives it to the next thread of equal priority (if one is available).

The job of the Java scheduler is to keep the highest priority thread running at all times, and if timeslicing is available, to ensure that several equally high-priority threads each execute for a quantum in round-robin fashion (i.e., these threads can be timesliced). Figure 15.2 illustrates Java’s multilevel priority queue for threads. In the figure, threads A and B each execute for a quantum in round-robin fashion until both threads complete execution. Next, thread C runs to completion. Then, threads D, E and F each execute for a quantum in round-robin fashion until they all complete execution. This process continues until all threads run to completion. Note that new higher-priority threads could postpone—possibly indefinitely—the execution of lower priority threads. Such \textit{indefinite postponement} often is referred to more colorfully as \textit{starvation}.

A thread’s priority can be adjusted with method \texttt{setPriority}, which takes an \texttt{int} argument. If the argument is not in the range 1 through 10, \texttt{setPriority} throws an \texttt{IllegalArgumentException}. Method \texttt{getPriority} returns the thread’s priority.

A thread can call the \texttt{yield} method to give other threads a chance to execute. Actually, whenever a higher priority thread becomes ready, the operating system preempts the current thread. So, a thread cannot \texttt{yield} to a higher priority thread, because the first thread would have been preempted when the higher priority thread became ready. Similarly, \texttt{yield} always allows the highest priority-ready thread to run, so if only lower priority threads are ready at the time of a \texttt{yield} call, the current thread will be the highest priority thread and will continue executing. Therefore, a thread \texttt{yields} to give threads of an equal priority a chance to run. On a timesliced system this is unnecessary, because threads of equal priority will each execute for their quantum (or until they lose the processor for some other reason), and other threads of equal priority will execute in round-robin fashion. Thus \texttt{yield} is appropriate for nontimesliced systems in which a thread would ordinarily run to completion before another thread of equal priority would have an opportunity to run.

\textbf{Performance Tip 15.3}

\textit{On nontimesliced systems, cooperating threads of equal priority should periodically call \texttt{yield} to enable their peers to proceed smoothly.}

\textbf{Portability Tip 15.2}

\textit{Java applets and applications should be programmed to work on all Java platforms to realize Java’s goal of true portability. When designing applets and applications that use threads, you must consider the threading capabilities of all the platforms on which the applets and applications will execute.}

A thread executes unless it dies, it becomes blocked by the operating system for input/output (or some other reason), it calls \texttt{sleep}, it calls \texttt{wait}, it calls \texttt{yield}, it is preempted by a thread of higher priority or its quantum expires. A thread with a higher priority than the running thread can become ready (and hence preempt the running thread) if a sleeping thread finishes sleeping, if I/O completes for a thread waiting for that I/O or if either \texttt{notify} or \texttt{notifyAll} is called on a thread that has called \texttt{wait}. 
The application of Fig. 15.3 demonstrates basic threading techniques, including creation of a class derived from `Thread`, construction of a `Thread` and using the `Thread` class `sleep` method. Each thread of execution we create in the program displays its name after sleeping for a random amount of time between 0 and 5 seconds. You will see that the main method (i.e., the main thread of execution) terminates before the application terminates. The program consists of two classes—`ThreadTester` (lines 4–28) and `PrintThread` (lines 33–71).

Class `PrintThread` inherits from `Thread`, so that each object of the class can execute in parallel. The class consists of instance variable `sleepTime`, a constructor and a
run method. Variable `sleepTime` stores a random integer value chosen when the program creates a `PrintThread` object. Each `PrintThread` object sleeps for the amount of time specified by its `sleepTime`, then outputs its name.

The `PrintThread` constructor (lines 38–48) initializes `sleepTime` to a random integer between 0 and 4999 (0 to 4.999 seconds). Then, the constructor outputs the name of the thread and the value of `sleepTime` to show the values for the particular `PrintThread` being constructed. The name of each thread is specified as a `String` argument to the `PrintThread` constructor and is passed to the superclass constructor at line 40.

[Note: It is possible to allow class `Thread` to choose a name for your thread by using the `Thread` class’s default constructor.]

```java
// Fig. 15.3: ThreadTester.java
// Show multiple threads printing at different intervals.

public class ThreadTester {

    // create and start threads
    public static void main( String args[] )
    {
        PrintThread thread1, thread2, thread3, thread4;

        // create four PrintThread objects
        thread1 = new PrintThread( "thread1" );
        thread2 = new PrintThread( "thread2" );
        thread3 = new PrintThread( "thread3" );
        thread4 = new PrintThread( "thread4" );

        System.err.println( "\nStarting threads" );

        // start executing PrintThreads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        System.err.println( "Threads started\n" );
    }
}
```

Fig. 15.3 Multiple threads printing at random intervals (part 1 of 3).
```java
public PrintThread(String name) {
    super(name);
    // sleep between 0 and 5 seconds
    sleepTime = (int) (Math.random() * 5000);
    // display name and sleepTime
    System.err.println("Name: " + getName() + "; sleep: " + sleepTime);
}

// control thread's execution
public void run() {
    // put thread to sleep for a random interval
    try {
        System.err.println(getName() + " going to sleep");
        // put thread to sleep
        Thread.sleep(sleepTime);
    }
    // if thread interrupted during sleep, catch exception
    // and display error message
    catch (InterruptedException interruptedException) {
        System.err.println(interruptedException.toString());
    }
    // print thread name
    System.err.println(getName() + " done sleeping");
}
} // end class PrintThread
```

```
Name: thread1; sleep: 3593
Name: thread2; sleep: 2653
Name: thread3; sleep: 4465
Name: thread4; sleep: 1318

Starting threads
Threads started

thread1 going to sleep
thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
thread4 done sleeping
thread2 done sleeping
thread1 done sleeping
thread3 done sleeping
```

Fig. 15.3 Multiple threads printing at random intervals (part 2 of 3).
When the program invokes a `PrintThread`'s `start` method (inherited from `Thread`), the `PrintThread` object enters the `ready` state. When the system assigns a processor to the `PrintThread` object, it enters the `running` state and its `run` method begins execution. Method `run` (lines 51–69) prints a `String` in the command window indicating that the thread is going to sleep (line 55), then invokes the `sleep` method (line 58) to place the thread into a `sleeping` state. At this point, the thread loses the processor, and the system allows another thread to execute. When the thread awakens, it is placed in a `ready` state again until the system assigns to the thread. When the `PrintThread` object enters the `running` state again, line 68 outputs the thread’s name (indicating that the thread is done sleeping), the `run` method terminates and the thread object enters the `dead` state. Note that method `sleep` can throw a checked `InterruptedException` (if another thread invokes the sleeping thread’s `interrupt` method); therefore, `sleep` must be called in a `try` block (in this example, we simply output the `String` representation of the exception if one occurs). Note that this example uses `System.err` rather than `System.out` to output lines of text. `System.err` represents the standard error object, which outputs error messages (normally to the command window). `System.out` performs buffered output—it is possible that a message output with `System.out` will not be output immediately. On the other hand, `System.err` uses unbuffered output—messages appear immediately when they are output. Using `System.err` in a multithreaded program helps ensure that the messages from our program are output in the correct order.

Class `ThreadTester`’s `main` method (lines 7–26) creates four objects of class `PrintThread` (lines 12–15) and invokes the `Thread` class `start` method on each one (lines 20–23) to place all four `PrintThread` objects in a `ready` state. Note that the program terminates execution when the last `PrintThread` awakens and prints its name. Also, note that the `main` method (i.e., the `main` thread of execution) terminates after starting the four `PrintThread` objects, but the application does not terminate until the last thread dies.
15.5 Thread Synchronization

Java uses monitors (as discussed by C.A.R. Hoare in his 1974 paper cited in Exercise 15.24) to perform synchronization. Every object with synchronized methods has a monitor. The monitor allows one thread at a time to execute a synchronized method on the object. This is accomplished by locking the object when the program invokes the synchronized method—also known as obtaining the lock. If there are several synchronized methods, only one synchronized method may be active on an object at once; all other threads attempting to invoke synchronized methods must wait. When a synchronized method finishes executing, the lock on the object is released and the monitor lets the highest priority-ready thread attempting to invoke a synchronized method proceed. [Note: Java also has synchronized blocks of code, which are discussed in the example of Section 15.10].

A thread executing in a synchronized method may determine that it cannot proceed, so the thread voluntarily calls wait. This removes the thread from contention for the processor and from contention for the monitor object. The thread now waits in the waiting state while other threads try to enter the monitor object. When a thread executing a synchronized method completes or satisfies the condition on which another thread may be waiting, the thread can notify a waiting thread to become ready again. At this point, the original thread can attempt to reacquire the lock on the monitor object and execute. The notify acts as a signal to the waiting thread that the condition for which the waiting thread has been waiting is now satisfied, so the waiting thread can reenter the monitor. If a thread calls notifyAll, then all threads waiting for the monitor become eligible to reenter the monitor (that is, they are all placed in a ready state). Remember that only one of those threads can obtain the lock on the object at a time—other threads that attempt to acquire the same lock will be blocked by the operating system until the lock becomes available again. Methods wait, notify and notifyAll are inherited by all classes from class Object. So any object may have a monitor.

Common Programming Error 15.1

Threads in the waiting state for a monitor object must be awakened explicitly with a notify (or interrupt) or the thread will wait forever. This may cause deadlock. [Note: There are versions of method wait that receive arguments indicating the maximum wait time. If the thread is not notified in the specified amount of time, the thread becomes ready to execute.]

Testing and Debugging Tip 15.3

Be sure that every call to wait has a corresponding call to notify that eventually will end the waiting or call notifyAll as a safeguard.

Performance Tip 15.4

Synchronization to achieve correct multithreaded behavior can make programs run more slowly due to the monitor overhead and frequently moving threads between the running, waiting and ready states. However, there is not much to say for highly efficient, incorrect multithreaded programs!

Testing and Debugging Tip 15.4

The locking that occurs with the execution of synchronized methods could lead to deadlock if the locks are never released. When exceptions occur, Java’s exception mechanism coordinates with Java’s synchronization mechanism to release appropriate synchronization locks to avoid these kinds of deadlocks.
Monitor objects maintain a list of all threads waiting to enter the monitor object to execute synchronized methods. A thread is inserted in the list and waits for the object if that thread calls a synchronized method of the object while another thread is already executing in a synchronized method of that object. A thread also is inserted in the list if the thread calls wait while operating inside the object. However, it is important to distinguish between waiting threads that blocked because the monitor was busy and threads that explicitly called wait inside the monitor. Upon completion of a synchronized method, outside threads that blocked because the monitor was busy can proceed to enter the object. Threads that explicitly invoked wait can proceed only when notified via a call by another thread to notify or notifyAll. When it is acceptable for a waiting thread to proceed, the scheduler selects the thread with the highest priority.

**Common Programming Error 15.2**

It is an error if a thread issues a wait, a notify or a notifyAll on an object without having acquired a lock for the object. This causes an IllegalMonitorStateException.

### 15.6 Producer/Consumer Relationship without Thread Synchronization

In a producer/consumer relationship, a producer thread calling a produce method may see that the consumer thread has not read the last message from a shared region of memory called a buffer, so the producer thread will call wait. When a consumer thread reads the message, it will call notify to allow a waiting producer to proceed. When a consumer thread enters the monitor and finds the buffer empty, it calls wait. A producer finding the buffer empty writes to the buffer, then calls notify so a waiting consumer can proceed.

Shared data can get corrupted if we do not synchronize access among multiple threads. Consider a producer/consumer relationship in which a producer thread deposits a sequence of numbers (we use 1, 2, 3, …) into a slot of shared memory. The consumer thread reads this data from the shared memory and prints it. We print what the producer produces as it produces it and what the consumer consumes as it consumes it. The program of Fig. 15.4–Fig. 15.7 demonstrates a producer and a consumer accessing a single shared cell of memory (int variable sharedInt in Fig. 15.6) without any synchronization. The threads are not synchronized, so data can be lost if the producer places new data into the slot before the consumer consumes the previous data. Also, data can be “doubled” if the consumer consumes data again before the producer produces the next item. To show these possibilities, the consumer thread in this example keeps a total of all the values it reads. The producer thread produces values from 1 to 10. If the consumer reads each value produced only once, the total would be 55. However, if you execute this program several times, you will see that the total is rarely, if ever, 55.

The program consists of four classes—ProduceInteger (Fig. 15.4), ConsumeInteger (Fig. 15.5), HoldIntegerUnsynchronized (Fig. 15.6) and SharedCell (Fig. 15.7).

Class ProduceInteger (Fig. 15.4)—a subclass of Thread—consists of instance variable sharedObject (line 4), a constructor (lines 7–11) and a run method (lines 15–37). The constructor initializes instance variable sharedObject (line 10) to refer to the
HoldIntegerUnsynchronized object shared, which was passed as an argument. Class ProduceInteger’s run method (lines 15–37) consists of a for structure that loops 10 times. Each iteration of the loop first invokes method sleep to put the ProduceInteger object into the sleeping state for a random time interval between 0 and 3 seconds. When the thread awakens, line 31 calls class HoldIntegerUnsynchronized’s setSharedInt method and passes the value of control variable count to set the shared object’s instance variable sharedInt. When the loop completes, lines 34–36 display a line of text in the command window indicating that the thread finished producing data and that the thread is terminating, then the thread terminates (i.e., the thread dies).

```java
public class ProduceInteger extends Thread {
    private HoldIntegerUnsynchronized sharedObject;

    public ProduceInteger( HoldIntegerUnsynchronized shared ) {
        super( "ProduceInteger" );
        sharedObject = shared;
    }

    public void run() {
        for ( int count = 1; count <= 10; count++ ) {
            // sleep for a random interval
            try {
                Thread.sleep( ( int ) ( Math.random() * 3000 ) );
            } catch( InterruptedException exception ) {
                System.err.println( exception.toString() );
            }
            // call sharedObject method from this
            // thread of execution
            sharedObject.setSharedInt( count );
            System.err.println( getName() + " finished producing values" +
                                "\nTerminating " + getName() );
        }
    }
}
```

Fig. 15.4 Class ProduceInteger represents the producer in a producer/consumer relationship.
Class **ConsumeInteger** (Fig. 15.5)—a subclass of **Thread**—consists of instance variable **sharedObject** (line 4), a constructor (lines 7–11) and a **run** method (lines 15–39). The constructor initializes instance variable **sharedObject** (line 10) to refer to the **HoldIntegerUnsynchronized** object **shared** that was passed as an argument. Class **ConsumeInteger**‘s **run** method (lines 15–39) consists of a **do/while** structure that loops until the value 10 is read from the **HoldIntegerUnsynchronized** object to which **sharedObject** refers. Each iteration of the loop invokes method **sleep** to put the **ConsumeInteger** object into the sleeping state for a random time interval between 0 and 3 seconds. Next, line 31 invokes class **HoldIntegerUnsynchronized**’s **getSharedInt** method to get the value of the shared object’s instance variable **sharedInt**. Then, line 32 adds the value returned by **getSharedInt** to the variable **sum**. When the loop completes, the **ConsumeInteger** thread displays a line in the command window indicating that it has finished consuming data and terminates (i.e., the thread dies).

```java
// Fig. 15.5: ConsumeInteger.java
// Definition of threaded class ConsumeInteger
public class ConsumeInteger extends Thread {
    private HoldIntegerUnsynchronized sharedObject;

    // initialize ConsumerInteger thread object
    public ConsumeInteger( HoldIntegerUnsynchronized shared )
    {
        super( "ConsumeInteger" );
        sharedObject = shared;
    }

    // ConsumeInteger thread loops until it receives 10
    // from sharedObject's getSharedInt method
    public void run()
    {
        int value, sum = 0;
        do {
            // sleep for a random interval
            try {
                Thread.sleep( (int) ( Math.random() * 3000 ) );
            }

            // process InterruptedException during sleep
            catch( InterruptedException exception ) {
                System.err.println( exception.toString() );
            }

            value = sharedObject.getSharedInt();
            sum += value;
        } while ( value != 10 );
    }
}
```

**Fig. 15.5** Class **ConsumeInteger** represents the consumer in a producer/consumer relationship (part 1 of 2).
Class `HoldIntegerUnsynchronized` consists of instance variable `sharedInt` (line 4), method `setSharedInt` (lines 7–13) and method `getSharedInt` (lines 16–22). Methods `setSharedInt` and `getSharedInt` do not synchronize access to instance variable `sharedInt`. Note that each method uses static `Thread` method `currentThread` to obtain a reference to the currently executing thread, then use `Thread` method `getName` to obtain the thread’s name.

Class `HoldIntegerUnsynchronized` maintains the data shared between the producer and consumer threads.

```java
36 System.err.println(
37     getName() + " retrieved values totaling: " + sum +
38     "\nTerminating " + getName() );
39 }
40 } // end class ConsumeInteger

Fig. 15.5 Class `ConsumeInteger` represents the consumer in a producer/consumer relationship (part 2 of 2).

Class `HoldIntegerUnsynchronized` represents the consumer in a producer/consumer relationship (part 2 of 2).
Ideally, we would like every value produced by the `ProduceInteger` object to be consumed exactly once by the `ConsumeInteger` object. However, when we study the output of Fig. 15.7, we see that the values 1, 3, 4, 6 and 7 are lost (i.e., never seen by the consumer) and that the values 8 and 9 are incorrectly retrieved more than once by the consumer.

```java
// Fig. 15.7: SharedCell.java
// Show multiple threads modifying shared object.
public class SharedCell {

    // execute application
    public static void main( String args[] )
    {
        HoldIntegerUnsynchronized sharedObject =
            new HoldIntegerUnsynchronized();

        // create threads
        ProduceInteger producer =
            new ProduceInteger( sharedObject );
        ConsumeInteger consumer =
            new ConsumeInteger( sharedObject );

        // start threads
        producer.start();
        consumer.start();
    }
}
```

**Fig. 15.7** Threads modifying a shared object without synchronization.

Ideally, we would like every value produced by the `ProduceInteger` object to be consumed exactly once by the `ConsumeInteger` object. However, when we study the output of Fig. 15.7, we see that the values 1, 3, 4, 6 and 7 are lost (i.e., never seen by the consumer) and that the values 8 and 9 are incorrectly retrieved more than once by the con-
sumer. Also, notice that the consumer twice retrieved value –1 (the default value of \texttt{sharedInt} set at line 5 of Fig. 15.6) before the producer ever assigned 1 to the \texttt{sharedInt} variable. This example clearly demonstrates that access to shared data by concurrent threads must be controlled carefully or a program may produce incorrect results.

To solve the problems of lost data and doubled data in the previous example, we will synchronize access of the concurrent producer and consumer threads to the shared data. Each method used by a producer or consumer to access the shared data is declared with the \texttt{synchronized} keyword. When a method declared \texttt{synchronized} is running in an object, the object is \textit{locked} so no other \texttt{synchronized} method can run in that object at the same time.

\section{15.7 Producer/Consumer Relationship with Thread Synchronization}

The application in Fig. 15.8 demonstrates a producer and a consumer accessing a shared cell of memory with synchronization so that the consumer only consumes after the producer produces a value. Classes \texttt{ProduceInteger} (Fig. 15.8), \texttt{ConsumeInteger} (Fig. 15.9) and \texttt{SharedCell} (Fig. 15.11) are identical to Fig. 15.4, Fig. 15.5 and Fig. 15.7 except that they use the new class \texttt{HoldIntegerSynchronized} in this example.

Class \texttt{HoldIntegerSynchronized} (Fig. 15.10) contains two instance variables—\texttt{sharedInt} (line 6) and \texttt{writeable} (line 7). Also, method \texttt{setSharedInt} (lines 12–39) and method \texttt{getSharedInt} (lines 44–70) are now \texttt{synchronized} methods. Objects of class \texttt{HoldIntegerSynchronized} have monitors, because \texttt{HoldIntegerSynchronized} contains \texttt{synchronized} methods. Instance variable \texttt{writeable} is known as the monitor’s \textit{condition variable}—is a \texttt{boolean} used by methods \texttt{setSharedInt} and \texttt{getSharedInt} of class \texttt{HoldIntegerSynchronized}. If \texttt{writeable} is \texttt{true}, \texttt{setSharedInt} can place a value into variable \texttt{sharedInt}, because the variable currently does not contain information. However, this means \texttt{getSharedInt} currently cannot read the value of \texttt{sharedInt}. If \texttt{writeable} is \texttt{false}, \texttt{getSharedInt} can read a value from variable \texttt{sharedInt} because the variable currently does contain information. However, this means \texttt{setSharedInt} currently cannot place a value into \texttt{sharedInt}.

\begin{lstlisting}[language=Java]
// Fig. 15.8: ProduceInteger.java
// Definition of threaded class ProduceInteger
public class ProduceInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;

    // initialize ProduceInteger thread object
    public ProduceInteger( HoldIntegerSynchronized shared )
    {
        super( "ProduceInteger" );
        sharedObject = shared;
    }
}
\end{lstlisting}

Fig. 15.8 Class \texttt{ProduceInteger} represents the producer in a producer/consumer relationship (part 1 of 2).
// ProduceInteger thread loops 10 times and calls
// sharedObject's setSharedInt method each time
public void run()
{
    for ( int count = 1; count <= 10; count++ ) {
        // sleep for a random interval
        try {
            Thread.sleep( ( int ) ( Math.random() * 3000 ) );
        } catch( InterruptedException exception ) {
            System.err.println( exception.toString() );
        }

        // call sharedObject method from this
        // thread of execution
        sharedObject.setSharedInt( count );
    }

    System.err.println( getName() + " finished producing values" +
                        "\nTerminating " + getName() );
} // end class ProduceInteger

Fig. 15.8  Class ProduceInteger represents the producer in a producer/
consumer relationship (part 2 of 2).

// Fig. 15.9: ConsumeInteger.java
// Definition of threaded class ConsumeInteger
public class ConsumeInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;

    // initialize ConsumerInteger thread object
    public ConsumeInteger( HoldIntegerSynchronized shared )
    {
        super( "ConsumeInteger" );
        sharedObject = shared;
    }

    // ConsumeInteger thread loops until it receives 10
    // from sharedObject's getSharedInt method
    public void run()
    {
        int value, sum = 0;

    }

    // Process InterruptedException during sleep
    catch( InterruptedException exception ) {
        System.err.println( exception.toString() );
    }
}

Fig. 15.9  Class ConsumeInteger represents the consumer in a producer/
consumer relationship (part 1 of 2).
When the `ProduceInteger` thread object invokes synchronized method `setSharedInt` (line 31 of Fig. 15.8), the thread acquires a lock on the `HoldIntegerSynchronized` monitor object. The `while` structure in `HoldIntegerSynchronized` at lines 14–25 tests the `writeable` variable with the condition `!writeable`. If this condition is true, the thread invokes method `wait`. This places the `ProduceInteger` thread object that called method `setSharedInt` into the waiting state for the `HoldIntegerSynchronized` object and releases the lock on it. Now another thread can invoke a synchronized method on the `HoldIntegerSynchronized` object.

The `ProduceInteger` object remains in the waiting state until it is notified that it may proceed—at which point it enters the ready state and waits for the system to assign a processor to it. When the `ProduceInteger` object reenters the running state, it reacquires the lock on the `HoldIntegerSynchronized` object implicitly and the `setSharedInt` method continues executing in the `while` structure with the next statement after `wait`. There are no more statements, so the program reevaluates the `while` condition. If the condition is false, the program outputs a line to the command window indicating that the producer is setting `sharedInt` to a new value, assigns `value` to `sharedInt`, sets `writeable` to false to indicate that the shared memory is now full (i.e., a consumer can read the value and a producer cannot put another value there yet) and invokes method `notify`. If there are any waiting threads, one of those waiting threads enters the ready state, indicating that the thread can now attempt its task again (as soon as it is assigned a processor). The `notify` method returns immediately and method `setSharedInt` returns to its caller.
// Fig. 15.10: HoldIntegerSynchronized.java
// Definition of class HoldIntegerSynchronized that
// uses thread synchronization to ensure that both
// threads access sharedInt at the proper times.
public class HoldIntegerSynchronized {
    private int sharedInt = -1;
    private boolean writeable = true;  // condition variable

    // synchronized method allows only one thread at a time to
    // invoke this method to set the value for a particular
    // HoldIntegerSynchronized object
    public synchronized void setSharedInt( int value )
    {
        while ( !writeable ) {  // not the producer's turn
            try {
                wait();
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }

        System.err.println( Thread.currentThread().getName() +
        " setting sharedInt to " + value );

        sharedInt = value;

        // indicate that producer cannot store another value until
        // a consumer retrieve current sharedInt value
        writeable = false;

        // tell a waiting thread to become ready
        notify();
    }

    // synchronized method allows only one thread at a time to
    // invoke this method to get the value for a particular
    // HoldIntegerSynchronized object
    public synchronized int getSharedInt()
    {
        while ( writeable ) {  // not the consumer's turn
            try {
                wait();
            }
        }

        // process Interrupted exception while thread waiting
        catch ( InterruptedException exception ) {
            exception.printStackTrace();
        }

        return sharedInt;
    }
}
Methods `getSharedInt` and `setSharedInt` are implemented similarly. When the `ConsumeInteger` object invokes method `getSharedInt`, the calling thread acquires a lock on the `HoldIntegerSynchronized` object. The `while` structure at lines 46–57 tests the `writeable` variable. If `writeable` is `true` (i.e., there is nothing to consume), the thread invokes method `wait`. This places the `ConsumeInteger` thread object that called method `getSharedInt` into the waiting state for the `HoldIntegerSynchronized` object and releases the lock on it so other synchronized methods can be invoked on the object. The `ConsumeInteger` object remains in the waiting state until it is `notified` that it may proceed—at which point it enters the `ready` state and waits to be assigned a processor. When the `ConsumeInteger` object reenters the running state, the thread reacquires the lock on the `HoldIntegerSynchronized` object and the `getSharedInt` method continues executing in the `while` structure with the next statement after `wait`. There are no more statements, so the program tests the `while` condition again. If the condition is `false`, the program sets `writeable` to `true` to indicate that the shared memory is now empty and invokes method `notify`. If there are any waiting threads, one of those waiting threads enters the `ready` state, indicating that the thread can now attempt its task again (as soon as it is assigned a processor). The `notify` method returns immediately. Then, `getSharedInt` outputs a line to the command window indicating that the consumer is retrieving `sharedInt`, then returns the value of `sharedInt` to `getSharedInt`’s caller.

Study the output in Fig. 15.11. Observe that every integer produced is consumed once—no values are lost and no values are doubled. Also, the consumer cannot read a value until the producer produces a value.
Fig. 15.11: Threads modifying a shared object with synchronization.
15.8 Producer/Consumer Relationship: The Circular Buffer

The program of Fig. 15.8–Fig. 15.11 does access the shared data correctly, but it may not perform optimally. Because the threads are running asynchronously, we cannot predict their relative speeds. If the producer wants to produce faster than the consumer can consume, it cannot do so. To enable the producer to continue producing, we can use a circular buffer that has enough extra cells to handle the “extra” production. The program of Fig. 15.13–Fig. 15.16 demonstrates a producer and a consumer accessing a circular buffer (in this case, a shared array of five cells) with synchronization so that the consumer only consumes a value when there are one or more values in the array, and the producer only produces a value when there are one or more available cells in the array. This program is implemented as a windowed application that sends its output to a `JTextArea`. Class `SharedCell`'s constructor creates the `HoldIntegerSynchronized`, `ProduceInteger` and `ConsumeInteger` objects. The `HoldIntegerSynchronized` object `sharedObject`'s constructor receives a reference to a `JTextArea` object in which the program’s output will appear.

This is the first program in which we use separate threads to modify the content displayed in Swing GUI components. The nature of multithreaded programming prevents the programmer from knowing exactly when a thread will execute. Swing components are not thread-safe—if multiple threads access a Swing GUI component, the results may not be correct. All interactions with Swing GUI components should be performed from one thread at a time. Normally, this thread is the event-dispatch thread (also known as the event-handling thread). Class `SwingUtilities` (package `javax.swing`) provides `static` method `invokeLater` to help with this process. Method `invokeLater` receives a `Runnable` argument. We will see in the next section that `Runnable` objects have a `run` method. In fact, class `Thread` implements interface `Runnable`, so all `Threads` have a `run` method. The threads in this example pass objects of class `UpdateThread` (Fig. 15.13) to method `invokeLater`. Each `UpdateThread` object receives a reference to the `JTextArea` in which to append the output and the message to display. Method `run` of class `UpdateThread` appends the message to the `JTextArea`. When the program calls `invokeLater`, the GUI component update will be queued for execution in the event-dispatch thread. The `run` method will then be invoked as part of the event-dispatch thread, ensuring that the GUI component updates in a thread-safe manner.

Common Programming Error 15.3

Interactions with Swing GUI components that change or obtain property values of the components may have incorrect results if the interactions are performed from multiple threads.

Software Engineering Observation 15.2

Use `SwingUtilities` static method `invokeLater` to ensure that all GUI interactions are performed from the event-dispatch thread.

Class `ProduceInteger` (Fig. 15.13) has been modified slightly from the version presented in Fig. 15.8. The new version places its output in a `JTextArea`. `ProduceInteger` receives a reference to the `JTextArea` when the program calls the `ProduceInteger` constructor. Note the use of `SwingUtilities` method `invokeLater` in lines 42–44 to ensure that the GUI updates properly.
// Fig. 15.12: UpdateThread.java
// Class for updating JTextArea with output.

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

public class UpdateThread extends Thread {
    private JTextArea outputArea;
    private String messageToOutput;

    // initialize outputArea and message
    public UpdateThread( JTextArea output, String message ) {
        outputArea = output;
        messageToOutput = message;
    }

    // method called to update outputArea
    public void run() {
        outputArea.append( messageToOutput );
    }
}

Fig. 15.12 UpdateThread used by SwingUtilities method invokeLater to ensure GUI updates properly.

// Fig. 15.13: ProduceInteger.java
// Definition of threaded class ProduceInteger

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

public class ProduceInteger extends Thread {
    private HoldIntegerSynchronized sharedObject;
    private JTextArea outputArea;

    // initialize ProduceInteger
    public ProduceInteger( HoldIntegerSynchronized shared,
        JTextArea output ) {
        super( "ProduceInteger" );
        sharedObject = shared;
        outputArea = output;
    }

    // ProduceInteger thread loops 10 times and calls sharedObject's setSharedInt method each time

Fig. 15.13 Class ProduceInteger represents the producer in a producer/consumer relationship (part 1 of 2).
```
23  public void run()
24  {
25      for ( int count = 1; count <= 10; count++ ) {
26        // sleep for a random interval
27        // Note: Interval shortened purposely to fill buffer
28        try {
29            Thread.sleep( (int) ( Math.random() * 500 ) );
30        }
31        
32        // process InterruptedException during sleep
33        catch( InterruptedException exception ) {
34            System.err.println( exception.toString() );
35        }
36        
37        sharedObject.setSharedInt( count );
38    }
39
40    // update Swing GUI component
41    SwingUtilities.invokeLater( new UpdateThread( outputArea,
42        "\n" + getName() + " finished producing values" +
43        "\nTerminating " + getName() + "\n" ) );
44
45  } // end class ProduceInteger
```

**Fig. 15.13** Class `ProduceInteger` represents the producer in a producer/consumer relationship (part 2 of 2).

Class `ConsumeInteger` (Fig. 15.14) has been modified slightly from the version presented in Fig. 15.9. The new version places its output in a `JTextArea`. `ConsumeInteger` receives a reference to the `JTextArea` when the program calls the `ConsumeInteger` constructor. Note the use of `SwingUtilities` method `invokeLater` in lines 45–47 to ensure that the GUI updates properly.

```
1  // Fig. 15.14: ConsumeInteger.java
2  // Definition of threaded class ConsumeInteger
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class ConsumeInteger extends Thread {
8      private HoldIntegerSynchronized sharedObject;
9      private JTextArea outputArea;
10
11    // initialize ConsumeInteger
12    public ConsumeInteger( HoldIntegerSynchronized shared,
13        JTextArea output )
14    {
15      super( "ConsumeInteger" );
```

**Fig. 15.14** Class `ConsumeInteger` represents the consumer in a producer/consumer relationship (part 1 of 2).
Once again, the primary changes in this example are in the definition of class `HoldIntegerSynchronized` (Fig. 15.15). The class now contains six instance variables—`sharedInt` is a five-element integer array that is used as the circular buffer, `writeable` indicates if a producer can write into the circular buffer, `readable` indicates if a consumer can read from the circular buffer, `readLocation` indicates the current position from which the consumer can read the next value, `writeLocation` indicates the next location in which the producer can place a value and `outputArea` is the `JTextArea` used by the threads in this program to display output.

Fig. 15.14 Class `ConsumeInteger` represents the consumer in a producer/consumer relationship (part 2 of 2).

```java
sharedObject = shared;
outputArea = output;
}

// ConsumeInteger thread loops until it receives 10
// from sharedObject's getSharedInt method
public void run()
{
    int value, sum = 0;
    do {
        // sleep for a random interval
        try {
            Thread.sleep( (int) ( Math.random() * 3000 ) );
        }
    } while ( value != 10 );

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea,
        "\n" + getName() + " retrieved values totaling: " +
        sum + "\nTerminating " + getName() + "\n" ) );
}

}  // end class ConsumeInteger
```

Fig. 15.15 Class `HoldIntegerSynchronized` monitors access to a shared array of integers (part 1 of 5).
// uses thread synchronization to ensure that both
// threads access sharedInt at the proper times.

// Java core packages
import java.text.DecimalFormat;

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

public class HoldIntegerSynchronized {

    // array of shared locations
    private int sharedInt[] = { -1, -1, -1, -1, -1 };

    // variables to maintain buffer information
    private boolean writeable = true;
    private boolean readable = false;
    private int readLocation = 0, writeLocation = 0;

    // GUI component to display output
    private JTextArea outputArea;

    // initialize HoldIntegerSynchronized
    public HoldIntegerSynchronized( JTextArea output )
    {
        outputArea = output;
    }

    // synchronized method allows only one thread at a time to
    // invoke this method to set a value in a particular
    // HoldIntegerSynchronized object
    public synchronized void setSharedInt( int value )
    {
        while ( !writeable ) {
            try {
                try {
                    // update Swing GUI component
                    SwingUtilities.invokeLater( new UpdateThread( outputArea, "WAITING TO PRODUCE " + value ) );
                    wait();
                } catch ( InterruptedException exception ) {
                    System.err.println( exception.toString() );
                }
            } finally {
                // process InterruptedException while thread waiting
                System.err.println( exception.toString() );
            }
        }
    }

    // Class HoldIntegerSynchronized monitors access to a shared array of integers (part 2 of 5).
// place value in writeLocation
sharedInt[ writeLocation ] = value;

// indicate that consumer can read a value
readable = true;

// update Swing GUI component
SwingUtilities.invokeLater( new UpdateThread( outputArea,
    "\nProduced " + value + " into cell " +
    writeLocation ) );

// update writeLocation for future write operation
writeLocation = ( writeLocation + 1 ) % 5;

// update Swing GUI component
SwingUtilities.invokeLater( new UpdateThread( outputArea,
    "\twrite " + writeLocation + "\tread " +
    readLocation ) );

displayBuffer( outputArea, sharedInt );

// test if buffer is full
if ( writeLocation == readLocation ) {
    writeable = false;

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea,
        "\nBUFFER FULL" ) );
}

// tell a waiting thread to become ready
notify();

}  // end method setSharedInt

// synchronized method allows only one thread at a time to
// invoke this method to get a value from a particular
// HoldIntegerSynchronized object
public synchronized int getSharedInt()
{
    int value;

    while ( !readable ) {
        try {

            // update Swing GUI component
            SwingUtilities.invokeLater( new UpdateThread( outputArea,
                " WAITING TO CONSUME" ) );

Fig. 15.15 Class HoldIntegerSynchronized monitors access to a shared array of integers (part 3 of 5).
```java
    wait();
}

// process InterruptedException while thread waiting
catch ( InterruptedException exception ) {
    System.err.println( exception.toString() );
}

// indicate that producer can write a value
writeable = true;

// obtain value at current readLocation
value = sharedInt[ readLocation ];

// update Swing GUI component
SwingUtilities.invokeLater( new UpdateThread( outputArea,
    \"Consumed \+ value \+ \" from cell \+ \
    readLocation ) );

// update read location for future read operation
readLocation = ( readLocation + 1 ) % 5;

// update Swing GUI component
SwingUtilities.invokeLater( new UpdateThread( outputArea,
    \"\+write \+ writeLocation \+ \"\+read \+ \
    readLocation ) );

displayBuffer( outputArea, sharedInt );

// test if buffer is empty
if ( readLocation == writeLocation ) {
    readable = false;

    // update Swing GUI component
    SwingUtilities.invokeLater( new UpdateThread( outputArea,
        \"\nBUFFER EMPTY\" ) );
}

// tell a waiting thread to become ready
notify();

return value;
}

// display contents of shared buffer
public void displayBuffer( JTextArea outputArea,
    int buffer[] ) {
    DecimalFormat formatNumber = new DecimalFormat( \" #;-#\" );
    StringBuffer outputBuffer = new StringBuffer();

    Fig. 15.15 Class HoldIntegerSynchronized monitors access to a shared array of integers (part 4 of 5).
Method `setSharedInt` (lines 34–87) performs the same tasks as it did in Fig. 15.10, with a few modifications. When execution continues at line 55 after the `while` loop, `setSharedInt` places the produced value in the circular buffer at location `writeLocation`. Next, `readable` is set to `true` (line 58), because there is at least one value in the buffer that the client can read. Lines 61–63 use `SwingUtilities` method `invokeLater` to append the value produced and the cell where the value was placed to the `JTextArea` (method `run` of class `UpdateThread` performs the actual append operation). Then, line 66 updates `writeLocation` for the next call to `setSharedInt`. The output continues with the current `writeLocation` and `readLocation` values and the values in the circular buffer (lines 69–73). If the `writeLocation` is equal to the `readLocation`, the circular buffer is currently full, so `writeable` is set to `false` (line 77) and the program displays the string `BUFFER FULL` (lines 80–81). Finally, line 85 invokes method `notify` to indicate that a waiting thread should move to the ready state.

Method `getSharedInt` (lines 92–149) also performs the same tasks in this example as it did in Fig. 15.10, with a few minor modifications. When execution continues at line 115 after the `while` loop, `getSharedInt` places the value at `readLocation` in the circular buffer. Lines 121–123 append to the `JTextArea` the value consumed and the cell from which the value was read. Then, line 126 updates `readLocation` for the next call to method `getSharedInt`. Lines 129–131 continue the output in the `JTextArea` with the current `writeLocation` and `readLocation` values and the current values in the circular buffer. If the `readLocation` is equal to the `writeLocation`, the circular buffer is currently empty, so `readable` is set to `false` (line 137), and lines 140–141 display the string `BUFFER EMPTY`. Finally, line 145 invokes method `notify` to place the next waiting thread into the `ready` state, and line 147 returns the retrieved value to the calling method.

In this version of the program, the outputs include the current `writeLocation` and `readLocation` values and the current contents of the buffer `sharedInt`. The elements of the `sharedInt` array were initialized to –1 for output purposes so that you can see each value inserted in the buffer. Notice that after the program places the fifth value in the fifth element of the buffer, the program inserts the sixth value at the beginning of the array—thus providing the `circular buffer` effect. Method `displayBuffer` (lines 152–166) uses
a `DecimalFormat` object to format the contents of the array `buffer`. The format control string " #;#-#" indicates a positive number format and a negative number format—the formats are separated by a semicolon (;). The format specifies that positive values should be preceded by a space and negative values should be preceded by a minus sign.

```java
// Fig. 15.16: SharedCell.java
// Show multiple threads modifying shared object.

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

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

public class SharedCell extends JFrame {
    // set up GUI
    public SharedCell()
    {
        super( "Demonstrating Thread Synchronization" );

        JTextArea outputArea = new JTextArea( 20, 30 );
        getContentPane().add( new JScrollPane( outputArea ) );

        setSize( 500, 500 );
        show();

        // set up threads
        HoldIntegerSynchronized sharedObject =
            new HoldIntegerSynchronized( outputArea );

        ProduceInteger producer =
            new ProduceInteger( sharedObject, outputArea );

        ConsumeInteger consumer =
            new ConsumeInteger( sharedObject, outputArea );

        // start threads
        producer.start();
        consumer.start();
    }

    // execute application
    public static void main( String args[] )
    {
        SharedCell application = new SharedCell();

        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}
```

Fig. 15.16 Threads modifying a shared array of cells (part 1 of 2).
15.9 Daemon Threads

A daemon thread is a thread that runs for the benefit of other threads. Unlike conventional user threads (i.e., any non-daemon thread in a program), daemon threads do not prevent a program from terminating. The garbage collector is a daemon thread. Nondaemon threads are conventional user threads or threads such as the event-dispatch thread used to process GUI events. We designate a thread as a daemon with the method call

```
setDaemon( true );
```

A `false` argument means that the thread is not a daemon thread. A program can include a mixture of daemon threads and nondaemon threads. When only daemon threads remain in a program, the program exits. If a thread is to be a daemon, it must be set as such before its `start` method is called or an `IllegalThreadStateException` is thrown. Method `isDaemon` returns `true` if a thread is a daemon thread and `false` otherwise.

**Common Programming Error 15.4**

Starting a thread, then attempting to make the thread a daemon thread, causes an `IllegalThreadStateException`.

Fig. 15.16 Threads modifying a shared array of cells (part 2 of 2).
Software Engineering Observation 15.3

The event-dispatch thread is an infinite loop and is not a daemon thread. As such, the event-dispatch thread will not terminate in a windowed application until the application calls System method exit.

Good Programming Practice 15.1

Do not assign critical tasks to a daemon thread. They are terminated without warning, which may prevent those tasks from completing properly.

15.10 Runnable Interface

Until now, we extended class Thread to create new classes that support multithreading. We overrode the run method to specify the tasks to be performed concurrently. However, if we want multithreading support in a class that already extends a class other than Thread, we must implement the Runnable interface in that class, because Java does not allow a class to extend more than one class at a time. Class Thread itself implements the Runnable interface (package java.lang) as expressed in the class header

```
public class Thread extends Object implements Runnable
```

Implementing the Runnable interface in a class enables a program to manipulate objects of that class as Runnable objects. As with deriving from the Thread class, the code that controls the thread is placed in method run.

A program that uses a Runnable object creates a Thread object and associates the Runnable object with that Thread. Class Thread provides four constructors that can receive references to Runnable objects as arguments. For example, the constructor

```
public Thread( Runnable runnableObject )
```

registers method run of runnableObject as the method to be invoked when the thread begins execution. The constructor

```
public Thread( Runnable runnableObject, String threadName )
```

constructs a Thread with the name threadName and registers method run of its runnableObject argument as the method to be invoked when the thread begins execution. As always, the thread object’s start method must be called to begin the thread’s execution.

Figure 15.17 demonstrates an applet with a private inner class and an anonymous inner class that each implement interface Runnable. The example also demonstrates how to suspend a thread (i.e., temporarily prevent it from executing), how to resume a suspended thread and how to terminate a thread that executes until a condition becomes false. Each of these techniques is important because Thread methods suspend, resume and stop were deprecated (i.e., they should no longer be used in Java programs) with the introduction of the Java 2 Platform. Because these methods are deprecated, we must code our own mechanisms for suspending, resuming and stopping threads. As we will demonstrate, these mechanisms rely on synchronized blocks of code, loops and boolean flag variables.
The applet class RandomCharacters displays three JLabels and three JCheckBoxes. A separate thread of execution is associated with each JLabel and button pair. Each thread randomly displays letters from the alphabet in its corresponding JLabel object. The applet defines the String alphabet (line 16) containing the letters from A to Z. This string is shared among the three threads. The applet’s start method (lines 52–65) instantiates three Thread objects (lines 59–60) and initializes each with an instance of class RunnableObject, which implements interface Runnable. Line 63 invokes the Thread class start method on each Thread, placing the threads in the ready state.

Class RunnableObject is defined at lines 115–185. Its run method (line 120) defines two local variables. Line 123 uses static method currentThread of class Thread to determine the currently executing Thread object. Line 125 calls the applet’s utility method getIndex (defined at lines 68–76) to determine the index of the currently executing thread in the array threads. The current thread displays a random character in the JLabel object with the same index in array outputs.

```java
// Fig. 15.17: RandomCharacters.java
// Demonstrating the Runnable interface.

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

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

public class RandomCharacters extends JApplet
    implements ActionListener {

    private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private final static int SIZE = 3;

    private JLabel outputs[];
    private JCheckBox checkboxes[];
    private Thread threads[];
    private boolean suspended[];

    public void init() {
        outputs = new JLabel[ SIZE ];
        checkboxes = new JCheckBox[ SIZE ];
        threads = new Thread[ SIZE ];
        suspended = new boolean[ SIZE ];

        Container container = getContentPane();
        container.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );
```

Fig. 15.17 Demonstrating the Runnable interface, suspending threads and resuming threads (part 1 of 5).
// create GUI components, register listeners and attach
// components to content pane
for ( int count = 0; count < SIZE; count++ ) {
    outputs[ count ] = new JLabel();
    outputs[ count ].setBackground( Color.green );
    outputs[ count ].setOpaque( true );
    container.add( outputs[ count ] );
}

checkboxes[ count ] = new JCheckBox( "Suspended" );
checkboxes[ count ].addActionListener( this );
container.add( checkboxes[ count ] );
}

// Create and start threads. This method called after init
// and when user revisits Web page containing this applet
public void start() {
    // create threads and start every time start is called
    for ( int count = 0; count < threads.length; count++ ) {
        // create Thread and initialize it with object that
        // implements Runnable
        threads[ count ] = new Thread( new RunnableObject(),
            "Thread " + ( count + 1 ) );

        // begin executing Thread
        threads[ count ].start();
    }
}

// determine thread location in threads array
private int getIndex( Thread current ) {
    for ( int count = 0; count < threads.length; count++ )
        if ( current == threads[ count ] )
            return count;
    return -1;
}

// called when user switches Web pages; stops all threads
public synchronized void stop() {
    // Indicate that each thread should terminate. Setting
    // these references to null causes each thread’s run
    // method to complete execution.
    for ( int count = 0; count < threads.length; count++ )
        threads[ count ] = null;
}

Fig. 15.17 Demonstrating the Runnable interface, suspending threads and
resuming threads (part 2 of 5).
// make all waiting threads ready to execute, so they
// can terminate themselves
notifyAll();
}

// handle button events
public synchronized void actionPerformed(ActionEvent event) {
    for (int count = 0; count < checkboxes.length; count++) {
        if (event.getSource() == checkboxes[count]) {
            suspended[count] = !suspended[count];
            // change label color on suspend/resume
            outputs[count].setBackground(!suspended[count] ? Color.green : Color.red);
            // if thread resumed, make sure it starts executing
            if (!suspended[count])
                notifyAll();
            return;
        }
    }
}

// private inner class that implements Runnable so objects
// of this class can control threads
private class RunnableObject implements Runnable {
    // Place random characters in GUI. Local variables
    // currentThread and index are declared final so
    // they can be used in an anonymous inner class.
    public void run() {
        // get reference to executing thread
        final Thread currentThread = Thread.currentThread();
        // determine thread's position in array
        final int index = getIndex(currentThread);
        // loop condition determines when thread should stop
        while (threads[index] == currentThread) {
            // sleep from 0 to 1 second
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                // Determine whether thread should suspend
                // execution. Use applet as monitor.
                synchronized(RandomCharacters.this) {

Fig. 15.17 Demonstrating the Runnable interface, suspending threads and
resuming threads (part 3 of 5).
while ( suspended[index] &&
        threads[index] == currentThread ) {
    // Temporarily stop thread execution. Use
    // applet as monitor.
    RandomCharacters.this.wait();
}

} // end synchronized block

// process InterruptedExceptions during sleep or wait
try ( InterruptedException interruptedException ) {
    System.err.println( "sleep interrupted" );
}

// display character on corresponding label
SwingUtilities.invokeLater(

    // anonymous inner class used by SwingUtilities
    // method invokeLater to ensure GUI
    // updates properly
    new Runnable() {

        // updates Swing GUI component
        public void run()
        {
            // pick random character
            char displayChar = alphabet.charAt(
                ( int ) ( Math.random() * 26 ) );

            outputs[index].setText(
                currentThread.getName() + ":: " +
                displayChar );
        }
    } // end anonymous inner class

); // end call to SwingUtilities.invokeLater

} // end while

System.err.println( 
    currentThread.getName() + " terminating" );

} } // end private inner class RunnableObject

} // end class RandomCharacters
The `while` loop at lines 129–179 continues to execute as long as the specified `Thread` reference is equal to the reference to the currently executing thread (`currentThread`). In each iteration of the loop, the thread sleeps for a random interval from 0 to 1 second.

When the user clicks the `JCheckBox` to the right of a particular `JLabel`, the corresponding `Thread` should be suspended (temporarily prevented from executing) or resumed (allowed to continue executing). In previous versions of Java, methods `suspend` and `resume` of class `Thread` were provided to suspend and resume a thread’s execution. These methods are now deprecated (i.e., they should no longer be used) because they introduce the possibility of deadlock in a program if they are not used correctly. Suspending and resuming of a thread can be implemented using thread synchronization and methods `wait` and `notify` of class `Object`. Lines 137–147 define a `synchronized` block of code (also called a `synchronized statement`) that helps suspend the currently executing `Thread`. When the `Thread` reaches the `synchronized` block, the applet object (referenced with `RandomCharacters.this`) is locked and the `while` structure tests `suspended[index]` to determine if the `Thread` should be suspended (i.e., `true`). If so, line 144 invokes method `wait` on the applet object to place the `Thread` in the waiting state. [Note the use of `RandomCharacters.this` to access the applet class’s `this` reference from the private inner class `RunnableObject`.] When the `Thread` should resume, the program tells all waiting threads to become ready to execute (we will discuss this shortly). However, only the resumed thread will get a chance to execute. The other suspended thread(s) will reenter the waiting state. Lines 156–177 use `SwingUtilities` method `invokeLater` to update the `JLabel` for the appropriate thread. This example uses an anonymous inner class to implement the `Runnable` interface (lines 161–175) and passes the anonymous inner class object to `invokeLater`. Lines 167–168 choose a random character from the `alphabet` string. Lines 170–172 display the character on the appropriate `JLabel` object.

Software Engineering Observation 15.4

An inner class can reference its outer class’s `this` reference by preceding the `this` reference with the outer class name and a dot operator.

If the user clicks the `Suspended` check box next to a particular `JLabel`, the program invokes method `actionPerformed` (lines 93–111). The method determines which checkbox received the event. Using the index of that checkbox in array `outputs`, line 98 toggles the corresponding `boolean` in array `suspended`. Lines 101–102 set the background color of the `JLabel` to red if the thread is being suspended and green if the thread...
is being resumed. If the appropriate boolean variable is false, the program calls method notifyAll (line 106) to move all waiting threads into the ready state and prepare them to resume execution. When each thread is dispatched to the processor to resume execution, the while condition at lines 139–140 in the run method fails for the resumed thread and the loop terminates. Execution of the run method then continues from line 156. For any other threads that became ready, but still are suspended, the condition at lines 139–140 remains true and the threads reenter the waiting state.

The applet’s stop method (lines 79–90) is provided to stop all three threads if the user leaves the Web page on which this applet resides (you can simulate this by selecting Stop from the appletviewer’s Applet menu). The for loop at lines 84–85 sets each Thread reference in array threads to null. Line 89 invokes Object method notifyAll to ensure that all waiting threads get ready to execute. When the program encounters the while loop condition at line 129 for each thread, the condition fails and the run method terminates. Thus, each thread dies. If the user returns to the Web page, the applet container calls the applet’s start method to instantiate and start three new threads.

**Performance Tip 15.5**

Stopping applet threads when leaving a Web page is a polite programming practice because it prevents your applet from using processor time (which can reduce performance) on the browser’s machine when the applet is not being viewed. The threads can be restarted from the applet’s start method, which is invoked by the browser when the Web page is revisited by the user.

### 15.11 Thread Groups

Sometimes it is useful to identify various threads as belonging to a thread group; class ThreadGroup contains methods for creating and manipulating thread groups. At constructor time, the group is given a unique name via a String argument.

The threads in a thread group can be manipulated as a group. It may, for example, be desirable to interrupt all the threads in a group. A thread group can be the parent thread group to a child thread group. Method calls sent to a parent thread group are also sent to all the threads in that parent’s child thread groups.

Class ThreadGroup provides two constructors. The constructor

```java
public ThreadGroup( String stringName )
```

constructs a ThreadGroup with name stringName. The constructor

```java
public ThreadGroup( ThreadGroup parentThreadGroup, String stringName )
```

constructs a child ThreadGroup of parentThreadGroup called stringName.

Class Thread provides three constructors that enable the programmer to instantiate a Thread and associate it with a ThreadGroup. The constructor

```java
public Thread( ThreadGroup threadGroup, String stringName )
```

constructs a Thread that belongs to threadGroup and has the name stringName. This constructor is normally invoked for derived classes of Thread whose objects should be associated with a ThreadGroup.
The constructor

```java
public Thread( ThreadGroup threadGroup,
               Runnable runnableObject )
```

constructs a `Thread` that belongs to `threadGroup` and that invokes the `run` method of `runnableObject` when the thread is assigned a processor to begin execution.

The constructor

```java
public Thread( ThreadGroup threadGroup,
               Runnable runnableObject, String stringName )
```

constructs a `Thread` that belongs to `threadGroup` and that invokes the `run` method of `runnableObject` when the thread is assigned a processor to begin execution. The name of this `Thread` is indicated by `stringName`.

Class `ThreadGroup` contains many methods for processing groups of threads. Some of these methods are summarized here. For more information on these methods, see the Java API documentation.

1. Method `activeCount` reports the number of active threads in a thread group plus the number of active threads in all its child thread groups.
2. Method `enumerate` has four versions. Two versions copy into an array of `Thread` references the active threads in the `ThreadGroup` (one of these also allows you to recursively get copies of all the active threads in child `ThreadGroup`). Two versions copy into an array of `ThreadGroup` references the active child thread groups in the `ThreadGroup` (one of these also allows you to recursively get copies of all the active thread groups in all the child `ThreadGroups`).
4. Method `getName` returns as a `String` the `ThreadGroup`’s name.
5. Method `getParent` determines the parent of a thread group.
6. Method `parentOf` returns `true` if the `ThreadGroup` to which the message is sent is the parent of, or the same as, the `ThreadGroup` supplied as an argument and returns `false` otherwise.

**Testing and Debugging Tip 15.5**

Method `list` lists the `ThreadGroup`. This can help in debugging.

15.12 (Optional Case Study) Thinking About Objects: Multithreading

Real-world objects perform their operations independently of one another and concurrently (in parallel). As you learned in this chapter, Java is a multithreaded programming language that facilitates the implementation of concurrent activities. The UML also contains support for designing concurrent models, as we will see shortly. In this section, we discuss how our simulation benefits from multithreading.
In “Thinking About Objects” Section 10.22, we encountered a problem with the collaboration diagram of Fig. 10.26—the waitingPassenger (the Person waiting to ride the Elevator) always enters the Elevator before the ridingPassenger (the Person riding the Elevator) exits. Proper use of multithreading in Java avoids this problem by guaranteeing that the waitingPassenger must wait for the ridingPassenger to exit the Elevator—as would happen in real life. In our collaboration diagrams, objects pass messages to other objects by calling methods of those other objects—a sending object may proceed only after a method returns. Java refers to such a message pass as a synchronous call. However, the synchronous call is not synchronized, because several objects may access the method at the same time—the synchronous call cannot guarantee exclusivity to one object. This poses a problem when we model our elevator simulation, because according to Fig. 10.26, the waitingPassenger and the ridingPassenger may occupy the Elevator at the same time, which violates the “capacity-of-one” requirement specified in the problem statement. In this section, we use a synchronized method to guarantee that only one Person may occupy the Elevator at a time.

**Threads, Active Classes and Synchronized Methods**
Java uses threads—flows of program control independent of other flows—to represent independent, concurrent activities. The UML provides the notion of an active class to represent a thread. Classes Elevator and Person are active classes (threads), because their objects must be able to operate concurrently and independently of one another and of other objects in the system. For example, the Elevator must be able to move between Floors while a Person is walking on a Floor. Figure 15.8 updates the collaboration diagram of Fig. 10.26 to support active classes, which are denoted by a thick black border in the diagram. To ensure that Persons enter and exit the Elevator in the proper order, we require a specific order in which to send messages—the Elevator must send message 3.3.1 (ridingPassenger exits) before sending message 3.2.1.1 (waitingPassenger enters the Elevator). The UML provides a notation to allow synchronization in collaboration diagrams—if we have two messages A and B, the notation B/A indicates that message A must wait for B to complete before message A occurs. For example, the 3.3.1/3.2.1.1 notation before the enterElevator message indicates that waitingPassenger must wait for ridingPassenger to exit the Elevator (message 3.3.1) before entering (message 3.2.1.1).

**Software Engineering Observation 15.5**
Messages in collaboration diagrams must complete in order (e.g., message 3.1 must complete before issuing message 3.2). However, messages between active classes may specify different ordering as necessary to guarantee synchronization between certain messages.

A Person must synchronize with the Elevator when traveling to guarantee that only one Person occupies the Elevator at a time. In Fig. 15.8, we include the ride message (3.2.1.2) to represent the Person riding the Elevator to the other Floor. The {concurrent} keyword placed after the ride message indicates that method ride is synchronized when we implement our design in Java. The Elevator contains method ride for the Person to allow the synchronization.
public synchronized void ride() {
    try {
        Thread.sleep( maxTravelTime );
    }
    catch ( InterruptedException interruptedException ) {
        // method doorOpened in Person interrupts method sleep;
        // Person has finished riding Elevator
    }
}
Method **ride** guarantees that only one **Person** may ride the **Elevator** at a time—as described in Section 15.5, only one **synchronized** method may be active on an object at once, so all other **Person** threads attempting to invoke **ride** must wait for the current thread to exit method **ride**.

Method **ride** invokes **static** method **sleep** of class **Thread** to put the **Person** thread into the **sleep** state, which represents the **Person** waiting for the ride to complete. We must specify the maximum amount of time that a **Person** will wait for the **Elevator** to complete traveling—however, there is no such information specified in the problem statement. We introduce a new attribute that represents this time—**maxTravelTime**, to which we arbitrarily assign a value of 10 minutes (i.e., a **Person** will wait 10 minutes for the travel to complete). Attribute **maxTravelTime** is a safeguard in case the **Elevator**—for whatever reason—never reaches the other **Floor**. The **Person** should never have to wait this long—if the **Person** waits 10 minutes, then the **Elevator** is broken, and we assume that our **Person** crawls out of the **Elevator** and exits the simulation.

**Software Engineering Observation 15.6**

*In a software-development process, the analysis phase yields a requirements document (e.g., our problem statement). As we continue the design and implementation phase, we discover additional issues that were not apparent to us at the analysis phase. As designers, we must anticipate these issues and deal with them accordingly.*

**Software Engineering Observation 15.7**

*One false assumption to make is that the system requirements remain stable (i.e., they provide all information necessary to build the system) throughout the analysis and design phases. In large systems that have long implementation phases, requirements can, and often do, change to accommodate those issues that were not apparent during analysis.*

If our **Elevator** works correctly, the **Elevator** travels for five seconds—specifically, invoking method **sleep** halts the **Elevator**’s thread for five seconds to simulate the travel. When the **Elevator** thread awakens, it sends **elevatorArrived** events as described in Section 10.22. The **elevatorDoor** receives this event and invokes method **doorOpened** (message **3.3**) of the **ridingPassenger**, as in:

```java
public void doorOpened( DoorEvent doorEvent )
{
    // set Person on Floor where Door opened
    setLocation( doorEvent.getLocation() );

    // interrupt Person's sleep method in run method and
    // Elevator's ride method
    interrupt();
}
```

Method **doorOpened** sets the **ridingPassenger**’s **Location** to the **Floor** at which the **Elevator** arrived, then calls the **ridingPassenger** thread’s **interrupt** method. The **interrupt** method terminates the **sleep** method invoked in method **ride**, method **ride** terminates, and the **ridingPassenger** leaves the **Elevator**, then exits the simulation. When the **ridingPassenger** exits the **Elevator**, the **ridingPassenger** releases the monitor on the **Elevator** object, which allows the **waitingPassenger** to invoke the **ride** method and obtain the monitor. Now, the **waitingPassenger** may invoke method **ride** to ride the **Elevator**.
Chapter 15 Multithreading

The *ridingPassenger* does not need to send the `exitElevator` message (message 3.3.1) to the *Elevator*, because the *waitingPassenger* cannot invoke `ride` until the *ridingPassenger* releases the *Elevator*’s monitor—the *ridingPassenger* releases the monitor when the *ridingPassenger* thread exits method `ride` after calling its `interrupt` method. Therefore, the *Person* thread’s `interrupt` method is equivalent to the `exitElevator` method (except that the *Person* sends itself the `interrupt` message), and we can substitute method `interrupt` for method `exitElevator`. Also, we can combine methods `ride` and `enterElevator` to handle both entering and riding the *Elevator*—our system needs only method `ride`, which allows a *Person* to obtain a monitor on the *Elevator*. As we implement our model in Appendix H, we use our collaboration diagram to help generate code in Java—however, we will make subtle “Java-specific” adjustments to our code to guarantee that the *Persons* enter and exit the *Elevator* correctly.

Sequence Diagrams

We now present the other type of interaction diagram, called a *sequence diagram*. Like the collaboration diagram, the sequence diagram shows interactions among objects; however, the sequence diagram emphasizes how messages are sent between objects over time. Both diagrams model interactions in a system. Collaboration diagrams emphasize what objects interact in a system, and sequence diagrams emphasize when these interactions occur.

Figure 15.9 is the sequence diagram for a *Person* changing floors. A rectangle enclosing the name of an object represents that object. We write object names in sequence diagrams using the same convention we have been using with collaboration diagrams. The dotted line extending down from an object’s rectangle is that object’s *lifeline*, which represents the progression of time. Actions occur along an object’s lifeline in chronological order from top to bottom—an action near the top of a lifeline happens before an action near the bottom. We provide several notes in this diagram to clarify where the *Person* exists in the simulation. Note that the diagram includes several dashed arrows. These arrows represent “return messages,” or the return of control to the sending object. Every message ultimately yields a return message. Showing return messages is not mandatory in a sequence diagram—we show return messages for clarity.

Message passing in sequence diagrams is similar to that in collaboration diagrams. An arrow extending from the object sending the message to the object receiving the message represents a message between two objects. The arrowhead points to the rectangle on the receiving object’s lifeline. As previously mentioned, when an object returns control, a return message—represented as a dashed line with an arrowhead—extends from the object returning control to the object that initially sent the message.

The sequence in Fig. 15.9 begins when a *Person* presses a *Button* on a *Floor* by sending message `pressButton` to that *Button*. The *Button* then requests the *Elevator* by sending message `requestElevator` to the *Elevator*.

The *Person* must wait for the *Elevator* to process this message before continuing. However, the *Person* does not need to wait for the *Elevator*’s arrival before proceeding with other actions. In our simulation, we force the *Person* to wait, but we could have had the *Person* perform other actions, such as read a newspaper, sway back and forth or place a call on a cell phone as the *Elevator* travels to the *Floor* of the *Person*. 
Note the split in flow for the Elevator after being requested; the flow of execution depends on which Floor generated the request. If the Elevator is on a different Floor.
than the request, the Elevator must move to the Floor on which the Person is waiting. To save space, the note informing that the Elevator moves to the other Floor represents this sequence (We will construct this sequence momentarily when we discuss how the Person rides the Elevator to the other Floor.) When the Elevator travels to the other Floor, the branched sequence merges with the original sequence in the Elevator’s lifeline, and the elevatorDoor sends an elevatorArrived message to the elevatorDoor upon arrival.

If the Elevator is on the same Floor as the request, the elevatorDoor immediately sends an elevatorArrived message to the elevatorDoor upon arrival. The elevatorDoor receives the arrival message and opens the Door on the arrival Floor. This door sends a doorOpened message to the Person—that Person then enters the Elevator.

The Person then presses the elevatorButton, which sends a button-Pressed event to the Elevator. The Elevator gets ready to leave by calling its private method setMoving, which changes the Elevator’s boolean attribute moving to true. The Elevator then closes the elevatorDoor, which closes the Door on that Floor. The Person then rides the Elevator by invoking the Elevator’s synchronized method ride. As in the collaboration diagram, the {concurrent} keyword placed after method ride indicates the method is synchronized, when implemented in Java.

The remainder of the diagram shows the sequence after the Elevator’s arrival at the destination Floor (also described in Fig. 15.9). Upon arrival, the Elevator stops moving by calling private method setMoving, which sets attribute moving to false. The Elevator then sends an elevatorArrived message to the elevatorDoor, which opens the Door on that Floor and sends the Person a doorOpened message. Method doorOpened wakes the sleeping Person’s thread by invoking method interrupt, causing method ride to terminate and return the Person’s thread to the “ready” state. The Person exits the simulation shortly afterwards. Note the large “X” at the bottom of the Person’s lifeline. In a sequence diagram, this “X” indicates that the associated object destroys itself (in Java, the Person object is marked for garbage collection).

**Our Final Class Diagram**

The integration of multithreading in our elevator model concludes the design of the model. We implement this model in Appendix H. Figure 15.10 presents the complete class diagram we use when implementing the model. Note that the major difference between the class diagrams of Fig. 15.10 and Fig. 10.31 is the presence of active classes in Fig. 15.10. We have established that classes Elevator and Person are active classes. However, the problem statement mentioned that “if a person neither enters nor requests the elevator, the elevator closes its door.” Having discussed multithreading, we believe that a better requirement would be for the Doors to close automatically (using a thread) if they have been open for more than a brief period (e.g., three seconds). In addition, although not mentioned in the problem statement, the Lights should turn off automatically—currently, the Lights turn off only when the Elevator departs from a Floor. We mark the Doors and Lights as active classes to handle this change. We implement this change in Appendix H.
Figure 15.11 presents the attributes and operations for all classes in Fig. 15.10. We use both diagrams to implement the model. We have omitted method enterElevator and exitElevator from class Elevator, because as discussed in the interaction diagrams, method ride appears to handle the Person’s entering, riding and exiting the Elevator. In addition, we have replaced method departElevator with private method setMoving, because according to Fig. 15.9, method setMoving provides the service that allows the Elevator to depart from a Floor. We also include private attribute maxTravelTime, which represents the maximum time the Person will wait
Chapter 15  Multithreading

Fig. 15.11 Final class diagram with attributes and operations.
15.13 (Optional) Discovering Design Patterns: Concurrent Design 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 object-oriented systems, such as concurrent, distributed or parallel systems. In this section, we discuss concurrency patterns to conclude our discussion of multithreaded programming.

Concurrency Design Patterns

Multithreaded programming languages such as Java allow designers to specify concurrent activities—that is, those that operate in parallel with one another. Designing concurrent systems improperly 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\(^1\) and Mark Grand\(^2\) created 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 (Grand, 98) prevents several threads from invoking the same method of another object concurrently. In Java, the synchronized keyword (discussed in Chapter 15) can be used to apply this pattern.

- The Guarded Suspension design pattern (Lea, 97) suspends a thread’s activity and resumes that thread’s activity when some condition is satisfied. Lines 137–147 and lines 95–109 of class RandomCharacters (Fig. 15.17) use this design pattern—methods wait and notify suspend and resume, respectively, the program threads, and line 98 toggles the variable that the condition evaluates.

- The Balking design pattern (Lea, 97) 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 (Lea, 97) 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 (Grand, 98) uses a two-phase termination process for a thread to ensure that a thread frees resources—such as other spawned threads—in memory (phase one) before termination (phase two).

---

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 “Discovering Design Patterns” Section 17.10, we return to the gang-of-four design patterns. Using the material introduced in Chapters 16 and 17, we identify those classes in package java.io and java.net that use design patterns.

**SUMMARY**

- Computers perform operations concurrently, such as compiling a program, printing a file and receiving e-mail messages over a network.
- Programming languages generally provide only a simple set of control structures that enable programmers to perform one action at a time, then proceed to the next action after the previous one is finished.
- The concurrency that computers perform today is normally implemented as operating system “primitives” available only to highly experienced “systems programmers.”
- Java makes concurrency primitives available to the programmer.
- Applications contain threads of execution, each thread designating a portion of a program that may execute concurrently with other threads. This capability is called multithreading.
- Java provides a low-priority garbage collector thread that reclaims dynamically allocated memory that is no longer needed. The garbage collector runs when processor time is available and there are no higher priority runnable threads. The garbage collector runs immediately when the system is out of memory to try to reclaim memory.
- Method run contains the code that controls a thread’s execution.
- A program launches a thread’s execution by calling the thread’s start method, which, in turn, calls method run.
- Method interrupt is called to interrupt a thread.
- Method isAlive returns true if start has been called for a given thread and the thread is not dead (i.e., the run method has not completed execution).
- Method setName sets the name of the Thread. Method getName returns the name of the Thread. Method toString returns a String consisting of the name of the thread, the priority of the thread and the thread’s group.
- Thread static method currentThread returns a reference to the executing Thread.
- Method join waits for the Thread on which join is called to die before the current Thread can proceed.
- Waiting can be dangerous; it can lead to two serious problems called deadlock and indefinite postponement. Indefinite postponement is also called starvation.
- A thread that was just created is in the born state. The thread remains in this state until the thread’s start method is called; this causes the thread to enter the ready state.
- A highest priority-ready thread enters the running state when the system assigns a processor to the thread.
- A thread enters the dead state when its run method completes or terminates for any reason. The system eventually will dispose of a dead thread.
• A running thread enters the blocked state when the thread issues an input/output request. A blocked thread becomes ready when the I/O it is waiting for completes. A blocked thread cannot use a processor even if one is available.

• When a running method calls wait, the thread enters a waiting state for the particular object in which the thread was running. A thread in the waiting state for a particular object becomes ready on a call to notify issued by another thread associated with that object.

• Every thread in the waiting state for a given object becomes ready on a call to notifyAll by another thread associated with that object.

• Every Java thread has a priority in the range Thread.MIN_PRIORITY (a constant of 1) and Thread.MAX_PRIORITY (a constant of 10). By default, each thread is given priority Thread.NORM_PRIORITY (a constant of 5).

• Some Java platforms support a concept called timeslicing and some do not. Without timeslicing, threads of equal priority run to completion before their peers get a chance to execute. With timeslicing, each thread receives a brief burst of processor time called a quantum during which that thread can execute. At the completion of the quantum, even if that thread has not finished executing, the processor is taken away from that thread and given to the next thread of equal priority, if one is available.

• The job of the Java scheduler is to keep a highest priority thread running at all times and, if timeslicing is available, to ensure that several equally high-priority threads each execute for a quantum in round-robin fashion.

• A thread’s priority can be adjusted with the setPriority method. Method getPriority returns the thread’s priority.

• A thread can call the yield method to give other threads a chance to execute.

• Every object that has synchronized methods has a monitor. The monitor lets only one thread at a time execute a synchronized method on the object.

• A thread executing in a synchronized method may determine that it cannot proceed, so the thread voluntarily calls wait. This removes the thread from contention for the processor and from contention for the object.

• A thread that has called wait is awakened by a thread that calls notify. The notify acts as a signal to the waiting thread that the condition the waiting thread has been waiting for is now (or could be) satisfied, so it is acceptable for that thread to reenter the monitor.

• A daemon thread serves other threads. When only daemon threads remain in a program, Java will exit. If a thread is to be a daemon, it must be set as such before its start method is called.

• To support multithreading in a class derived from some class other than Thread, implement the Runnable interface in that class.

• Implementing the Runnable interface gives us the ability to treat the new class as a Runnable object (just like inheriting from a class allows us to treat our subclass as an object of its superclass). As with deriving from the Thread class, the code that controls the thread is placed in the run method.

• A thread with a Runnable class is created by passing to the Thread class constructor a reference to an object of the class that implements the Runnable interface. The Thread constructor registers the run method of the Runnable object as the method to be invoked when the thread begins execution.

• Class ThreadGroup contains the methods for creating and manipulating groups of related threads in a program.
Chapter 15  Multithreading

TERMINOLOGY

asynchronous threads
blocked (state of a thread)
blocked on I/O
busy wait
child thread group
circular buffer
concurrent execution of threads
condition variable
consumer
consumer thread
currentThread method
daemon thread
dead (state of a thread)
deadlock
destroy method
dumpStack method
Error class ThreadDeath
execution context
fixed-priority scheduling
garbage collection by a low-priority thread
getName method
getParent method of ThreadGroup class
highest priority runnable thread
I/O completion
IllegalArgumentException
IllegalMonitorStateException
IllegalThreadStateException
indefinite postponement
inherit thread priority
interrupt method
interrupted method
InterruptedException
InterruptedException class
interthread communication
isAlive method
isDaemon method
isInterrupted method
join method
kill a thread
MAX_PRIORITY(10)
memory leak
MIN_PRIORITY(1)
monitor
multiple inheritance
multiprocessing
multithreaded program
multithreaded server
multithreading
new (state of a thread)
nonpreemptive scheduling
NORM_PRIORITY(5)
notify method
notifyAll method
parallelism
parent thread
parent thread group
preemptive scheduling
priority of a thread
producer thread
producer/consumer relationship
programmer-defined thread
quantum
round-robin scheduling
run method
Runnable interface (in java.lang package)
runnable state (of a thread)
running (thread state)
scheduler
scheduling a thread
setDaemon method
setName method
setPriority method
shared objects
single-threaded language
single-threaded program
sleep method of Thread class
sleeping state (of a thread)
start method
starvation
synchronization
synchronized method
thread
Thread class (in java.lang package)
thread group
thread priority
thread safe
thread states
thread synchronization
Thread.MAX_PRIORITY
Thread.MIN_PRIORITY
Thread.NORM_PRIORITY
Thread.sleep()
ThreadDeath exception
ThreadGroup class
timeslicing
wait method
yield method
SELF-REVIEW EXERCISES

15.1 Fill in the blanks in each of the following statements:

a) C and C++ are ________-threaded languages whereas Java is a ________-threaded language.
b) Java provides a ________ thread that reclaims dynamically allocated memory.
c) Java eliminates most ________ errors that occur commonly in languages like C and C++ when dynamically allocated memory is not explicitly reclaimed by the program.
d) Three reasons a thread that is alive could be not runnable (i.e., blocked) are ________, ________, and ________.
e) A thread enters the dead state when ________.
f) A thread’s priority can be changed with the ________ method.
g) A thread may give up the processor to a thread of the same priority by calling the method.
h) To wait for a designated number of milliseconds and resume execution, a thread should call the ________ method.
i) The ________ method moves a thread in the object’s waiting state to the ready state.

15.2 State whether each of the following is true or false. If false, explain why.

a) A thread is not runnable if it is dead.
b) In Java, a higher priority runnable thread will preempt threads of lower priority.
c) The Windows and Windows NT Java systems use timeslicing. Therefore, they can enable threads to preempt threads of the same priority.
d) Threads may yield to threads of lower priority.

ANSWERS TO SELF-REVIEW EXERCISES

15.1 a) single, multi.  b) garbage collector.  c) memory leak.  d) waiting, sleeping, blocked for input/output.  e) its run method terminates.  f) setPriority.  g) yield.  h) sleep.  i) notify.

15.2 a) True.  b) True.  c) False. Timeslicing allows a thread to execute until its timeslice (or quantum) expires. Then other threads of equal priority can execute.  d) False. Threads can only yield to threads of equal priority.

EXERCISES

15.3 State whether each of the following is true or false. If false, explain why.

a) The sleep method does not consume processor time while a thread sleeps.
b) Declaring a method synchronized guarantees that deadlock cannot occur.
c) Java provides a powerful capability called multiple inheritance.
d) Thread methods suspend and resume are deprecated.

15.4 Define each of the following terms.

a) thread
b) multithreading
c) ready state
d) blocked state
e) preemptive scheduling
f) Runnable interface
g) monitor
h) notify method
i) producer/consumer relationship

15.5 a) List each of the reasons stated in this chapter for using multithreading.
b) List additional reasons for using multithreading.
15.6 List each of the three reasons given in the text for entering the blocked state. For each of these, describe how the program will normally leave the blocked state and enter the runnable state.

15.7 Distinguish between preemptive scheduling and nonpreemptive scheduling. Which does Java use?

15.8 What is timeslicing? Give a fundamental difference in how scheduling is performed on Java systems that support timeslicing vs. scheduling on Java systems that do not support timeslicing.

15.9 Why would a thread ever want to call \texttt{yield}?

15.10 What aspects of developing Java applets for the World Wide Web encourage applet designers to use \texttt{yield} and \texttt{sleep} abundantly?

15.11 If you choose to write your own \texttt{start} method, what must you be sure to do to ensure that your threads start up properly?

15.12 Distinguish among each of the following means of pausing threads:
   a) busy wait
   b) sleep
   c) blocking I/O

15.13 Write a Java statement that tests if a thread is alive.

15.14 a) What is multiple inheritance?
    b) Explain why Java does not offer multiple inheritance.
    c) What feature does Java offer instead of multiple inheritance?
    d) Explain the typical use of this feature.
    e) How does this feature differ from \texttt{abstract} classes?

15.15 Distinguish between the notions of \texttt{extends} and \texttt{implements}.

15.16 Discuss each of the following terms in the context of monitors:
   a) monitor
   b) producer
   c) consumer
   d) \texttt{wait}
   e) \texttt{notify}
   f) \texttt{InterruptedException}
   g) \texttt{synchronized}

15.17 (Tortoise and the Hare) In the Chapter 7 exercises, you were asked to simulate the legendary race of the tortoise and the hare. Implement a new version of that simulation, this time placing each of the animals in a separate thread. At the start of the race call the \texttt{start} methods for each of the threads. Use \texttt{wait}, \texttt{notify} and \texttt{notifyAll} to synchronize the animals’ activities.

15.18 (Multithreaded, Networked, Collaborative Applications) In Chapter 17, we cover networking in Java. A multithreaded Java application can communicate concurrently with several host computers. This creates the possibility of being able to build some interesting kinds of collaborative applications. In anticipation of studying networking in Chapter 17, develop proposals for several possible multi-threaded networked applications. After studying Chapter 17, implement some of those applications.

15.19 Write a Java program to demonstrate that as a high-priority thread executes, it will delay the execution of all lower priority threads.

15.20 If your system supports timeslicing, write a Java program that demonstrates timeslicing among several equal-priority threads. Show that a lower priority thread’s execution is deferred by the timeslicing of the higher-priority threads.

15.21 Write a Java program that demonstrates a high priority thread using \texttt{sleep} to give lower priority threads a chance to run.
If your system does not support timeslicing, write a Java program that demonstrates two threads using `yield` to enable one another to execute.

Two problems that can occur in systems like Java, that allow threads to wait, are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in a multithreaded Java program.


a) With multithreading, many threads can access shared data; as we have seen, access to shared data needs to be synchronized carefully to avoid corrupting the data.

b) Consider an airline-reservation system in which many clients are attempting to book seats on particular flights between particular cities. All of the information about flights and seats is stored in a common database in memory. The database consists of many entries, each representing a seat on a particular flight for a particular day between particular cities. In a typical airline-reservation scenario, the client will probe around in the database looking for the “optimal” flight to meet that client’s needs. So a client may probe the database many times before deciding to book a particular flight. A seat that was available during this probing phase could easily be booked by someone else before the client has a chance to book it. In that case, when the client attempts to make the reservation, the client will discover that the data has changed and the flight is no longer available.

c) The client probing around the database is called a reader. The client attempting to book the flight is called a writer. Clearly, any number of readers can be probing shared data at once, but each writer needs exclusive access to the shared data to prevent the data from being corrupted.

d) Write a multithreaded Java program that launches multiple reader threads and multiple writer threads, each attempting to access a single reservation record. A writer thread has two possible transactions, `makeReservation` and `cancelReservation`. A reader has one possible transaction, `queryReservation`.

e) First implement a version of your program that allows unsynchronized access to the reservation record. Show how the integrity of the database can be corrupted. Next implement a version of your program that uses Java monitor synchronization with `wait` and `notify` to enforce a disciplined protocol for readers and writers accessing the shared reservation data. In particular, your program should allow multiple readers to access the shared data simultaneously when no writer is active. But if a writer is active, then no readers should be allowed to access the shared data.

f) Be careful. This problem has many subtleties. For example, what happens when there are several active readers and a writer wants to write? If we allow a steady stream of readers to arrive and share the data, they could indefinitely postpone the writer (who may become tired of waiting and take his or her business elsewhere). To solve this problem, you might decide to favor writers over readers. But here, too, there is a trap, because a steady stream of writers could then indefinitely postpone the waiting readers, and they, too, might choose to take their business elsewhere! Implement your monitor with the following methods: `startReading`, which is called by any reader who wants to begin accessing
a reservation; \texttt{stopReading} to be called by any reader who has finished reading a reservation; \texttt{startWriting} to be called by any writer who wants to make a reservation and \texttt{stopWriting} to be called by any writer who has finished making a reservation.

15.25 Write a program that bounces a blue ball inside an applet. The ball should be initiated with a \texttt{mousePressed} event. When the ball hits the edge of the applet, the ball should bounce off the edge and continue in the opposite direction.

15.26 Modify the program of Exercise 15.25 to add a new ball each time the user clicks the mouse. Provide for a minimum of 20 balls. Randomly choose the color for each new ball.

15.27 Modify the program of Exercise 15.26 to add shadows. As a ball moves, draw a solid black oval at the bottom of the applet. You may consider adding a 3-D effect by increasing or decreasing the size of each ball when a ball hits the edge of the applet.

15.28 Modify the program of Exercise 15.25 or 15.26 to bounce the balls off each other when they collide.
16

Files and Streams

Objectives

• To be able to create, read, write and update files.
• To understand the Java streams class hierarchy.
• To be able to use the FileInputStream and FileOutputStream classes.
• To be able to use the ObjectInputStream and ObjectOutputStream classes.
• To be able to use class RandomAccessFile.
• To be able to use a JFileChooser dialog to access files and directories.
• To become familiar with sequential-access and random-access file processing.
• To be able to use class File.

I can only assume that a “Do Not File” document is filed in a “Do Not File” file.

Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975

Consciousness … does not appear to itself chopped up in bits. … A “river” or a “stream” are the metaphors by which it is most naturally described.

William James

I read part of it all the way through.

Samuel Goldwyn

It is quite a three-pipe problem.

Sir Arthur Conan Doyle
16.1 Introduction

Storage of data in variables and arrays is temporary—the data is lost when a local variable “goes out of scope” or when the program terminates. Programs use files for long-term retention of large amounts of data, even after programs that create the data terminate. We refer to data maintained in files as persistent data, because the data exists beyond the duration of program execution. Computers store files on secondary storage devices such as magnetic disks, optical disks, and magnetic tapes. In this chapter, we explain how Java programs create, update, and process data files. We consider both “sequential-access” files and “random-access” files and discuss typical applications for each.

File processing is one of the most important capabilities a language must have to support commercial applications that typically process massive amounts of persistent data. In this chapter, we discuss Java’s powerful file-processing and stream input/output features. File processing is a subset of Java’s stream-processing capabilities that enable a program to read and write bytes in memory, in files, and over network connections. We have two goals in this chapter—to introduce file-processing paradigms and to provide the reader with sufficient stream-processing capabilities to support the networking features introduced in Chapter 17.

Software Engineering Observation 16.1

It would be dangerous to enable applets arriving from anywhere on the World Wide Web to be able to read and write files on the client system. By default, Web browsers prevent applets from performing file processing on the client system. Therefore, file-processing programs generally are implemented as Java applications.

16.2 Data Hierarchy

Ultimately, a computer processes all data items as combinations of zeros and ones, because it is simple and economical for engineers to build electronic devices that can assume two stable states—one state represents 0, the other state represents 1. It is remarkable that the
impressive functions performed by computers involve only the most fundamental manipulations of 0s and 1s.

The smallest data item in a computer can assume the value 0 or the value 1. Such a data item is called a bit (short for “binary digit”—a digit that can assume one of two values). Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from 1 to 0 or from 0 to 1).

It is cumbersome for programmers to work with data in the low-level form of bits. Instead, programmers prefer to work with data in such forms as decimal digits (0–9), letters (A–Z and a–z), and special symbols (e.g., $, @, %, *, (, -, +, ,, ;, ?, / and many others). Digits, letters and special symbols are known as characters. The computer’s character set is the set of all characters used to write programs and represent data items. Computers can process only 1s and 0s, so a computer’s character set represents every character as a pattern of 1s and 0s. Characters in Java are Unicode characters composed of 2 bytes. Bytes are most commonly composed of eight bits. Programmers create programs and data items with characters. Computers manipulate and process these characters as patterns of bits. See Appendix K for more information on Unicode.

Just as characters are composed of bits, fields are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters can be used to represent a person’s name.

Data items processed by computers form a data hierarchy in which data items become larger and more complex in structure as we progress from bits, to characters, to fields, etc.

Typically, several fields (called instance variables in Java) compose a record (implemented as a class in Java). In a payroll system, for example, a record for a particular employee might consist of the following fields (possible data types for these fields are shown in parentheses following each field):

- Employee identification number (int)
- Name (String)
- Address (String)
- Hourly pay rate (double)
- Number of exemptions claimed (int)
- Year-to-date earnings (int or double)
- Amount of taxes withheld (int or double)

Thus, a record is a group of related fields. In the preceding example, each of the fields belongs to the same employee. Of course, a particular company might have many employees and will have a payroll record for each employee. A file is a group of related records. A company’s payroll file normally contains one record for each employee. Thus, a payroll file for a small company might contain only 22 records, whereas a payroll file for a large company might contain 100,000 records. It is not unusual for a company to have many files, some containing millions, or even billions, of characters of information. Figure 16.1 illustrates the data hierarchy.

1. More generally, a file can contain arbitrary data in arbitrary formats. In some operating systems, a file is viewed as nothing more than a collection of bytes. In such an operating system, any organization of the bytes in a file (such as organizing the data into records) is a view created by the applications programmer.
Chapter 16  
Files and Streams  

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key. A record key identifies a record as belonging to a particular person or entity that is unique from all other records. In the payroll record described previously, the employee identification number normally would be chosen as the record key.

There are many ways to organize records in a file. The most common organization is called a sequential file in which records are stored in order by the record-key field. In a payroll file, records are placed in order by employee identification number. The first employee record in the file contains the lowest employee identification number, and subsequent records contain increasingly higher employee identification numbers.

Most businesses store data in many different files. For example, companies might have payroll files, accounts receivable files (listing money due from clients), accounts payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other file types. Often, a group of related files is called a database. A collection of programs designed to create and manage databases is called a database management system (DBMS).

16.3 Files and Streams

Java views each file as a sequential stream of bytes (Fig. 16.2). Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained administrative data structure. Java abstracts this concept from the programmer. A Java program processing a stream of bytes simply receives an indication from the system when the program reaches the end of the stream—the program does not need to know how the underli-
ing platform represents files or streams. In some cases, the end-of-file indication occurs as an exception. In other cases, the indication is a return value from a method invoked on a stream-processing object. We demonstrate both cases in this chapter.

A Java program opens a file by creating an object and associating a stream of bytes with the object. Java also can associates streams of bytes associated with devices. In fact, Java creates three stream objects that are associated with devices when a Java program begins executing—System.in, System.out and System.err. The streams associated with these objects provide communication channels between a program and a particular device. For example, object System.in (the standard input stream object) normally enables a program to input bytes from the keyboard, object System.out (the standard output stream object) normally enables a program to output data to the screen and object System.err (the standard error stream object) normally enables a program to output error messages to the screen. Each of these streams can be redirected. For System.in, this enables the program to read bytes from a different source. For System.out and System.err, this enables the output to be sent to a different location, such as a file on disk. Class System provides methods setIn, setOut and setError to redirect the standard input, output and error streams.

Java programs perform file processing by using classes from package java.io. This package includes definitions for the stream classes, such as FileInputStream (for byte-based input from a file), FileOutputStream (for byte-based output to a file), FileReader (for character-based input from a file) and FileWriter (for character-based output to a file). Files are opened by creating objects of these stream classes that inherit from classes InputStream, OutputStream, Reader and Writer, respectively. Thus, the methods of these stream classes can all be applied to file streams as well. To perform input and output of data types, objects of class ObjectInputStream, DataInputStream, ObjectOutputStream and DataOutputStream will be used together with the byte-based file stream classes FileInputStream and FileOutputStream. Figure 16.3 summarizes the inheritance relationships of many of the Java I/O classes (abstract classes are shown in italic font). The following discussion overviews the capabilities of each of the classes in Fig. 16.3.

Java offers many classes for performing input/output. This section briefly overviews many of these classes and explains how they relate to one another. In the rest of the chapter, we use several of these stream classes as we implement a variety of file-processing programs that create, manipulate and destroy sequential-access files and random-access files. We also include a detailed example on class File, which is useful for obtaining information about files and directories. In Chapter 17, Networking, we use stream classes extensively to implement networking applications.

![Fig. 16.2](https://example.com/fig162.png) Java’s view of a file of n bytes.
Fig. 16.3  A portion of the class hierarchy of the java.io package (part 1 of 2).
InputStream and OutputStream (subclasses of Object) are abstract classes that define methods for performing byte-based input and output, respectively.

Programs perform byte-based file input/output with FileInputStream (a subclass of InputStream) and FileOutputStream (a subclass of OutputStream). We use these classes extensively in the examples in this chapter.

Pipes are synchronized communication channels between threads or processes. Java provides PipedOutputStream (a subclass of OutputStream) and PipedInputStream (a subclass of InputStream) to establish pipes between two threads. One thread sends data to another by writing to a PipedOutputStream. The target thread reads information from the pipe via a PipedInputStream.

A PrintStream (a subclass of FilterOutputStream) performs text output to the specified stream. Actually, we have been using PrintStream output throughout the text to this point—System.out is a PrintStream, as is System.err.

A FilterInputStream filters an InputStream; filtering simply means that the filter stream provides additional functionality, such as buffering, monitoring line numbers or aggregating data bytes into meaningful primitive-data-type units. FilterInputStream and FilterOutputStream are abstract classes, so additional functionality is provided by their subclasses.

Reading data as raw bytes is fast but crude. Usually programs read data as aggregates of bytes that form an int, a float, a double and so on. Java programs can use several classes to input and output data in aggregate form.

A RandomAccessFile is useful for direct-access applications, such as transaction-processing applications like airline-reservations systems and point-of-sale systems. With a sequential-access file, each successive input/output request reads or writes the next consecutive set of data in the file. With a random-access file, each successive input/output request could be directed to any part of the file—perhaps one widely separated from the part of the file referenced in the previous request. Direct-access applications provide rapid access to specific data items in large files; often, such applications are used in applications that require users to wait for answers—these answers must be made available quickly, or the people might become impatient and “take their business elsewhere.”
The `DataInput` interface is implemented by class `DataInputStream` and class `RandomAccessFile` (discussed later in the chapter); each needs to read primitive data types from a stream. `DataInputStream` enable a program to read binary data from an `InputStream`. The `DataInput` interface includes methods `read` (for `byte` arrays), `readBoolean`, `readByte`, `readChar`, `readDouble`, `readFloat`, `readFully` (for `byte` arrays), `readInt`, `readLong`, `readShort`, `readUnsignedByte`, `readUnsignedShort`, `readUTF` (for strings) and `skipBytes`.

The `DataOutput` interface is implemented by class `DataOutputStream` (a subclass of `FilterOutputStream`) and class `RandomAccessFile`; each needs to write primitive data types to an `OutputStream`. `DataOutputStream` enable a program to write binary data to an `OutputStream`. The `DataOutput` interface includes methods `flush`, `size`, `write` (for a `byte`), `write` (for a `byte` array), `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars` (for Unicode Strings), `writeDouble`, `writeFloat`, `writeInt`, `writeLong`, `writeShort` and `writeUTF`.

Buffering is an I/O-performance-enhancement technique. With a `BufferedOutputStream` (a subclass of class `FilterOutputStream`), each output statement does not necessarily result in an actual physical transfer of data to the output device. Rather, each output operation is directed to a region in memory called a buffer that is large enough to hold the data of many output operations. Then, actual transfer to the output device is performed in one large physical output operation each time the buffer fills. The output operations directed to the output buffer in memory are often called logical output operations. With a `BufferedOutputStream`, a partially filled buffer can be forced out to the device at any time by invoking the stream object’s `flush` method.

**Performance Tip 16.1**

Since typical physical output operations are extremely slow compared to the speed of accessing computer memory, buffered outputs normally yield significant performance improvements over unbuffered outputs.

With a `BufferedInputStream` (a subclass of class `FilterInputStream`), many “logical” chunks of data from a file are read as one large physical input operation into a memory buffer. As a program requests each new chunk of data, it is taken from the buffer (this is sometimes referred to as a logical input operation). When the buffer is empty, the next actual physical input operation from the input device is performed to read in the next group of “logical” chunks of data. Thus, the number of actual physical input operations is small compared with the number of read requests issued by the program.

**Performance Tip 16.2**

Since typical input operations are extremely slow compared to the speed of accessing computer memory, buffered inputs normally yield significant performance improvements over unbuffered inputs.

A `PushbackInputStream` (a subclass of class `FilterInputStream`) is for applications more exotic than those most programmers require. Essentially, the application reading a `PushbackInputStream` reads bytes from the stream and forms aggregates consisting of several bytes. Sometimes, to determine that one aggregate is complete, the application must read the first character “past the end” of the first aggregate. Once the program determines that the current aggregate is complete, the extra character is “pushed
back” onto the stream. **PushbackInputStream**s are used by programs (like compilers) that parse their inputs—that is, break them into meaningful units (such as the keywords, identifiers and operators that the compiler must recognize).

When object instance variables are output to a disk file, in a sense we lose the object’s type information. We have only data, not type information, on a disk. If the program that is going to read this data knows what object type it corresponds to, then the data is simply read into objects of that type. Sometimes, we would like to read or write an entire object to a file. The **ObjectInputStream** and **ObjectOutputStream** classes, which respectively implement the **ObjectInput** and **ObjectOutput** interfaces, enable an entire object to be read from or written to a file (or other stream type). We often chain **ObjectInputStreams** to **FileInputStreams**. (We also chain **ObjectOutputStreams** to **FileOutputStreams**.) The **ObjectOutput** interface contains method **writeObject**, which takes an **Object** that implements interface **Serializable** as an argument and writes its information to the **OutputStream**. Correspondingly, the **ObjectInput** interface requires method **readObject**, which reads and returns an **Object** from an **InputStream**. After the reading of an object, it can be cast to the desired type. Additionally, these interfaces include other **Object**-centric methods as well as the same methods as **DataInput** and **DataOutput** for reading and writing primitive data types.

Java stream I/O includes capabilities for inputting from **byte** arrays in memory and outputting to **byte** arrays in memory. A **ByteArrayInputStream** (a subclass of **InputStream**) reads from a **byte** array in memory. A **ByteArrayOutputStream** (a subclass of **OutputStream**) outputs to a **byte** array in memory. One application of **byte**-array I/O is data validation. A program can input an entire line at a time from the input stream into a **byte** array. Then, a validation routine can scrutinize the contents of the **byte** array and correct the data, if necessary. Then, the program can proceed to input from the **byte** array, knowing that the input data is in the proper format. Outputting to a **byte** array is a nice way to take advantage of the powerful output-formatting capabilities of Java streams. For example, data can be prepared in a **byte** array, using the same formatting that will be displayed at a later time, then output to a disk file to preserve the screen image.

A **SequenceInputStream** (a subclass of **InputStream**) enables concatenation of several **InputStreams**, so that the program sees the group as one continuous **InputStream**. As the program reaches the end of an input stream, that stream closes and the next stream in the sequence opens.

In addition to the byte based streams, Java provides **Reader** and **Writer** classes, which are Unicode, two-byte, character based streams. Most of the byte-based streams have corresponding character-based **Reader** or **Writer** classes.

Class **BufferedReader** (a subclass of abstract class **Reader**) and class **BufferedWriter** (a subclass of abstract class **Writer**) enable efficient buffering for character-based streams. Character-based streams use Unicode characters—such streams can process data in any language that the Unicode character set represents.

Class **CharArrayReader** and class **CharArrayWriter** read and write a stream of characters to a character array.

A **PushbackReader** (a subclass of abstract class **FilterReader**) enables characters to be pushed back on a character stream. A **LineReader** (a subclass of **BufferedReader**) is a buffered character-stream that keeps track of line numbers (i.e., a newline, a return or a carriage-return line-feed combination).
Class **FileReader** (a subclass of **InputStreamReader**) and class **FileWriter** (a subclass of **OutputStreamWriter**) read characters from and write characters to a file, respectively. Class **PipedReader** and class **PipedWriter** implement piped-character streams that can be used to transfer information between threads. Class **StringReader** and **StringWriter** read and write characters to **Strings**. A **PrintWriter** writes characters to a stream.

Class **File** enables programs to obtain information about a file or directory. We discuss class **File** extensively in Section 16.12.

### 16.4 Creating a Sequential-Access File

Java imposes no structure on a file. Notions like “record” do not exist in Java files. Therefore, the programmer must structure files to meet the requirements of applications. In the following example, we see how the programmer can impose a simple record structure on a file. First we present the program, then we analyze it in detail.

The program of Fig. 16.4–Fig. 16.6 creates a simple sequential-access file that might be used in an accounts receivable system to help manage the money owed by a company’s credit clients. For each client, the program obtains an account number, the client’s first name, the client’s last name and the client’s balance (i.e., the amount the client still owes the company for goods and services received in the past). The data obtained for each client constitutes a record for that client. The program uses the account number as the record key; that is, the file will be created and maintained in account-number order. *[Note: This program assumes the user enters the records in account-number order. In a comprehensive accounts receivable system, a sorting capability would be provided so the user could enter the records in any order—the records would then be sorted and written to the file.]*

Most of the programs in this chapter have a similar GUI, so this program defines class **BankUI** (Fig. 16.4) to encapsulate the GUI. (See the second sample output screen in Fig. 16.6.) Also, the program defines class **AccountRecord** (Fig. 16.5) to encapsulate the client record information (i.e., account, first name, etc.) used by the examples in this chapter. For reuse, classes **BankUI** and **AccountRecord** are defined in package `com.deitel.jhtp4.ch16`.

*Note: Most of the programs in this chapter use classes **BankUI** and **AccountRecord**. When you compile these classes, or any others that will be reused in this chapter, you should place the classes in a common directory. When you compile classes that use **BankUI** and **AccountRecord**, be sure to specify the **-classpath** command line argument to both **javac** and **java**, as in*

```
javac -classpath .;packageLocation  ClassName.java
java -classpath .;packageLocation  ClassName
```

where **packageLocation** represents the common directory in which the classes of the package `com.deitel.jhtp4.ch16` reside and **ClassName** represents the class to compile or execute. Be sure to include the current directory (specified with .) in the class path. *[Note: If your packaged classes are in a JAR file, the **packageLocation** should include the location and name of the actual JAR file.]* Also, the path separator shown (.; which is used in Microsoft Windows) should be appropriate for your platform (such as : on UNIX/Linux].*

Class **BankUI** (Fig. 16.4) contains two **JButtons** and arrays of **JLabels** and **JTextFields**. The number of **JLabels** and **JTextFields** is set with the constructor
defined at lines 34–75. Methods `getFieldValues` (lines 116–124), `setFieldValues` (lines 104–113) and `clearFields` (lines 96–100) manipulate the text of the `JTextField`s. Methods `getFields` (lines 90–93), `getDoTask1Button` (lines 78–81) and `getDoTask2Button` (lines 84–87) return individual GUI components, so that a client program can add `ActionListeners` (for example).

```java
// Fig. 16.4: BankUI.java
// A reusable GUI for the examples in this chapter.
package com.deitel.jhtp4.ch16;

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

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

public class BankUI extends JPanel {

    // label text for GUI
    protected final static String names[] = { "Account number",
        "First name", "Last name", "Balance",
        "Transaction Amount" };

    // GUI components; protected for future subclass access
    protected JLabel labels[];
    protected JTextField fields[];
    protected JButton doTask1, doTask2;
    protected JPanel innerPanelCenter, innerPanelSouth;

    // number of text fields in GUI
    protected int size;

    // constants representing text fields in GUI
    public static final int ACCOUNT = 0, FIRSTNAME = 1,
        LASTNAME = 2, BALANCE = 3, TRANSACTION = 4;

    // Set up GUI. Constructor argument of 4 creates four rows
    // of GUI components. Constructor argument of 5 (used in a
    // later program) creates five rows of GUI components.
    public BankUI( int mySize )
    {
        size = mySize;
        labels = new JLabel[ size ];
        fields = new JTextField[ size ];

        // create labels
        for ( int count = 0; count < labels.length; count++ )
            labels[ count ] = new JLabel( names[ count ] );

        // create text fields
        for ( int count = 0; count < fields.length; count++ )
            fields[ count ] = new JTextField();
```

Fig. 16.4  `BankUI` contains a reusable GUI for several programs (part 1 of 3).
// create panel to lay out labels and fields
innerPanelCenter = new JPanel();
innerPanelCenter.setLayout( new GridLayout( size, 2 ) );

// attach labels and fields to innerPanelCenter
for ( int count = 0; count < size; count++ ) {
    innerPanelCenter.add( labels[ count ] );
    innerPanelCenter.add( fields[ count ] );
}

// create generic buttons; no labels or event handlers
doTask1 = new JButton();
doTask2 = new JButton();

// create panel to lay out buttons and attach buttons
innerPanelSouth = new JPanel();
innerPanelSouth.add( doTask1 );
innerPanelSouth.add( doTask2 );

// set layout of this container and attach panels to it
setLayout( new BorderLayout() );
add( innerPanelCenter, BorderLayout.CENTER );
add( innerPanelSouth, BorderLayout.SOUTH );

// validate layout
validate();

} // end constructor

// return reference to generic task button doTask1
public JButton getDoTask1Button()
{
    return doTask1;
}

// return reference to generic task button doTask2
public JButton getDoTask2Button()
{
    return doTask2;
}

// return reference to fields array of JTextFields
public JTextField[] getFields()
{
    return fields;
}

// clear content of text fields
public void clearFields()
{
    for ( int count = 0; count < size; count++ )
        fields[ count ].setText( "" );

Fig. 16.4  BankUI contains a reusable GUI for several programs (part 2 of 3).
Class \texttt{AccountRecord} (Fig. 16.5) implements interface \texttt{Serializable}, which allows objects of \texttt{AccountRecord} to be used with \texttt{ObjectInputStream}s and \texttt{ObjectOutputStream}s. Interface \texttt{Serializable} is known as a \textit{tagging interface}. Such an interface contains no methods. A class that implements this interface is \textit{tagged} as being a \texttt{Serializable} object, which is important because an \texttt{ObjectOutputStream} will not output an object unless it is a \texttt{Serializable} object. In a class that implements \texttt{Serializable}, the programmer must ensure that every instance variable of the class is a \texttt{Serializable} type, or must declare particular instance variables as \texttt{transient} to indicate that those variables are not \texttt{Serializable} and they should be ignored during the serialization process. By default, all primitive type variables are transient. For non-primitive types, you must check the definition of the class (and possibly its superclasses) to ensure that the type is \texttt{Serializable}. Class \texttt{AccountRecord} contains \texttt{private} data members \texttt{account}, \texttt{firstName}, \texttt{lastName} and \texttt{balance}. This class also provides \texttt{public} “get” and “set” methods for accessing the \texttt{private} data members.

```java
// Fig. 16.5: AccountRecord.java
// A class that represents one record of information.
package com.deitel.jhtp4.ch16;
```
// Java core packages
import java.io.Serializable;

public class AccountRecord implements Serializable {
    private int account;
    private String firstName;
    private String lastName;
    private double balance;

    // no-argument constructor calls other constructor with
    // default values
    public AccountRecord()
    {
        this( 0, "", "", 0.0 );
    }

    // initialize a record
    public AccountRecord( int acct, String first, String last, double bal )
    {
        setAccount( acct );
        setFirstName( first );
        setLastName( last );
        setBalance( bal );
    }

    // set account number
    public void setAccount( int acct )
    {
        account = acct;
    }

    // get account number
    public int getAccount()
    {
        return account;
    }

    // set first name
    public void setFirstName( String first )
    {
        firstName = first;
    }

    // get first name
    public String getFirstName()
    {
        return firstName;
    }
}

Fig. 16.5 Class AccountRecord maintains information for one account (part 2 of 3).
Now, let us discuss the code that creates the sequential-access file (Fig. 16.6). In this example, we introduce class JFileChooser (package javax.swing) for selecting files (as on the second screen in Fig. 16.6). Line 103 constructs a JFileChooser instance and assigns it to reference fileChooser. Lines 104–105 call method setFileSelectionMode to specify what the user can select from the fileChooser. For this program, we use JFileChooser static constant FILES_ONLY to indicate that only files can be selected. Other static constants include FILES_AND_DIRECTORIES and DIRECTORIES_ONLY.

```java
// set last name
public void setLastName( String last )
{
    lastName = last;
}

// get last name
public String getLastName()
{
    return lastName;
}

// set balance
public void setBalance( double bal )
{
    balance = bal;
}

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

} // end class AccountRecord
```

Fig. 16.5 Class AccountRecord maintains information for one account (part 3 of 3).

Fig. 16.6 Creating a sequential file (part 1 of 6).
import com.deitel.jhtp4.ch16.BankUI;
import com.deitel.jhtp4.ch16.AccountRecord;

public class CreateSequentialFile extends JFrame {
    private ObjectOutputStream output;
    private BankUI userInterface;
    private JButton enterButton, openButton;

    // set up GUI
    public CreateSequentialFile()
    {
        super( "Creating a Sequential File of Objects" );

        // create instance of reusable user interface
        userInterface = new BankUI( 4 ); // four textfields
        getContentPane().add( userInterface, BorderLayout.CENTER );

        // get reference to generic task button doTask1 in BankUI
        // and configure button for use in this program
        openButton = userInterface.getDoTask1Button();
        openButton.setText( "Save into File ..." );

        // register listener to call openFile when button pressed
        openButton.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent event ) {
                openFile();
            }
        } );

        // get reference to generic task button doTask2 in BankUI
        // and configure button for use in this program
        enterButton = userInterface.getDoTask2Button();
        enterButton.setText( "Enter" );
        enterButton.setEnabled( false ); // disable button

        // register listener to call addRecord when button pressed
        enterButton.addActionListener( new ActionListener() {
            // anonymous inner class to handle enterButton event
        } );

    }


// call addRecord when button pressed
public void actionPerformed( ActionEvent event )
{
    addRecord();
}
}  // end anonymous inner class

); // end call to addActionListener

// register window listener to handle window closing event
addWindowListener(
    // anonymous inner class to handle windowClosing event
    new WindowAdapter() {
        // add current record in GUI to file, then close file
        public void windowClosing( WindowEvent event )
        {
            if ( output != null )
                addRecord();
            closeFile();
        }
    }  // end anonymous inner class
); // end call to addWindowListener

setSize( 300, 200 );
show();
}  // end CreateSequentialFile constructor

// allow user to specify file name
private void openFile()
{
    // display file dialog, so user can choose file to open
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
    int result = fileChooser.showSaveDialog( this );
    if ( result == JFileChooser.CANCEL_OPTION )
        return;
    // get selected file
    File fileName = fileChooser.getSelectedFile();
// display error if invalid
if ( fileName == null ||
    fileName.getName().equals( "" ) )
    JOptionPane.showMessageDialog( this,
        "Invalid File Name", "Invalid File Name",
        JOptionPane.ERROR_MESSAGE );

else {
    // open file
    try {
        output = new ObjectOutputStream(
            new FileOutputStream( fileName ) );
        openButton.setEnabled( false );
        enterButton.setEnabled( true );
    } catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error Opening File", "Error",
            JOptionPane.ERROR_MESSAGE );
    }
}

// process exceptions from opening file
catch ( IOException ioException ) {
    JOptionPane.showMessageDialog( this,
        "Error Opening File", "Error",
        JOptionPane.ERROR_MESSAGE );
}

} // end method openFile

// close file and terminate application
private void closeFile()
{
    // close file
    try {
        output.close();
        System.exit( 0 );
    }

    // process exceptions from closing file
    catch( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error closing file", "Error",
            JOptionPane.ERROR_MESSAGE );
        System.exit( 1 );
    }

    // add record to file
    public void addRecord()
    {
        int accountNumber = 0;
        AccountRecord record;
        String fieldValues[] = userInterface.getFieldValues();

Fig. 16.6 Creating a sequential file (part 4 of 6).
// if account field value is not empty
if ( ! fieldValues[ BankUI.ACCOUNT ].equals( "" ) ) {

  // output values to file
  try {
    accountNumber = Integer.parseInt(
      fieldValues[ BankUI.ACCOUNT ] );
    if ( accountNumber > 0 ) {

      // create new record
      record = new AccountRecord( accountNumber,
        fieldValues[ BankUI.FIRSTNAME ],
        fieldValues[ BankUI.LASTNAME ],
        Double.parseDouble(
          fieldValues[ BankUI.BALANCE ] ) );

      // output record and flush buffer
      output.writeObject( record );
      output.flush();
    }

    // clear textfields
    userInterface.clearFields();
  }

  // process invalid account number or balance format
  catch ( NumberFormatException formatException ) {
    JOptionPane.showMessageDialog( this,
      "Bad account number or balance",
      "Invalid Number Format",
      JOptionPane.ERROR_MESSAGE );
  }

  // process exceptions from file output
  catch ( IOException ioException ) {
    closeFile();
  }

  } // end if

} // end method addRecord

// execute application; CreateSequentialFile constructor
// displays window
public static void main( String args[] )
{
  new CreateSequentialFile();
}

Fig. 16.6 Creating a sequential file (part 5 of 6).
Line 107 calls method `showSaveDialog` to display the `JFileChooser` dialog titled `Save`. Argument `this` specifies the `JFileChooser` dialog’s `parent` window, which determines the position of the dialog on the screen. If `null` is passed, the dialog is displayed in the center of the window; otherwise, the dialog is centered over the application window. When displayed, a `JFileChooser` dialog does not allow the user to interact with any other program window until the user closes the `JFileChooser` dialog by clicking `Save` or `Cancel`. Dialogs that behave in this fashion are called modal dialogs. The user selects the drive, directory and file name, then clicks `Save`. Method `showSaveDialog` returns an integer specifying which button the user clicked (`Save` or `Cancel`) to close the dialog. Line 110 tests whether the user clicked `Cancel` by comparing `result` to static constant `CANCEL_OPTION`. If so, the method returns.
Line 114 retrieves the file the user selected by calling method `getSelectedFile`, which returns an object of type `File` that encapsulates information about the file (e.g., name and location), but does not represent the contents of the file. This `File` object does not open the file. We assign this `File` object to the reference `fileName`.

As stated previously, a program opens a file by creating an object of stream class `FileInputStream` or `FileOutputStream`. In this example, the file is to be opened for output, so the program creates a `FileOutputStream`. One argument is passed to the `FileOutputStream`'s constructor—a `File` object. Existing files opened for output are truncated—all data in the file is discarded.

**Common Programming Error 16.1**

It is a logic error to open an existing file for output when, in fact, the user wants to preserve the file. The contents of the file are discarded without warning.

Class `FileOutputStream` provides methods for writing `byte` arrays and individual `bytes` to a file. For this program, we need to write objects to a file—a capability not provided by `FileOutputStream`. The solution to this problem is a technique called *chaining of stream objects*—the ability to add the services of one stream to another. To chain an `ObjectOutputStream` to the `FileOutputStream`, we pass the `FileOutputStream` object to the `ObjectOutputStream`'s constructor (lines 127–128). The constructor could throw an `IOException` if a problem occurs during opening of the file (e.g., when a file is opened for writing on a drive with insufficient space, a read-only file is opened for writing or a nonexistent file is opened for reading). If so, the program displays a `JOptionPane`. If construction of the two streams does not throw an `IOException`, the file is open. Then, reference `output` can be used to write objects to the file.

The program assumes data is input correctly and in the proper record number order. The user populates the `JTextField`s and clicks **Enter** to write the data to the file. The **Enter** button's `actionPerformed` method (lines 66–69) calls our method `addRecord` (lines 164–211) to perform the write operation. Line 188 calls method `writeObject` to write the `record` object to file. Line 189 calls method `flush` to ensure that any data stored in memory is written to the file immediately.

When the user clicks the close box (the **X** in the window’s top-right corner), the program calls method `windowClosing` (lines 82–88), which compares `output` to `null` (line 84). If `output` is not `null`, the stream is open and methods `addRecord` and `closeFile` (lines 145–161) are called. Method `closeFile` calls method `close` for `output` to close the file.

**Performance Tip 16.3**

Always release resources explicitly and at the earliest possible moment at which it is determined that the resource is no longer needed. This makes the resource immediately available to be reused by your program or by another program, thus improving resource utilization.

When using chained stream objects, the outermost object (the `ObjectOutputStream` in this example) should be used to close the file.

**Performance Tip 16.4**

Explicitly close each file as soon as it is known that the program will not reference the file again. This can reduce resource usage in a program that will continue executing long after it no longer needs to be referencing a particular file. This practice also improves program clarity.
In the sample execution for the program of Fig. 16.6, we entered information for five accounts (see Fig. 16.7). The program does not show how the data records actually appear in the file. To verify that the file has been created successfully, in the next section we create a program to read the file.

### 16.5 Reading Data from a Sequential-Access File

Data are stored in files so that they may be retrieved for processing when needed. The previous section demonstrated how to create a file for sequential access. In this section, we discuss how to read data sequentially from a file.

The program of Fig. 16.8 reads records from a file created by the program of Fig. 16.6 and displays the contents of the records. The program opens the file for input by creating a `FileInputStream` object. The program specifies the name of the file to open as an argument to the `FileInputStream` constructor. In Fig. 16.6, we wrote objects to the file, using an `ObjectOutputStream` object. Data must be read from the file in the same format in which it was written to the file. Therefore, we use an `ObjectInputStream` chained to a `FileInputStream` in this program. Note that the third sample screen capture shows the GUI displaying the last record in the file.

**Sample Data**

<table>
<thead>
<tr>
<th>Account</th>
<th>Name</th>
<th>Account</th>
<th>Name</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Bob</td>
<td>Jones</td>
<td>24.98</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>Steve</td>
<td>Doe</td>
<td>-345.67</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>Pam</td>
<td>White</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>Sam</td>
<td>Stone</td>
<td>-42.16</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>Sue</td>
<td>Rich</td>
<td>224.62</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 16.7 Sample data for the program of Fig. 16.6.

```java
// Fig. 16.8: ReadSequentialFile.java
// This program reads a file of objects sequentially
// and displays each record.

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

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

// Deitel packages
import com.deitel.jhtp4.ch16.*;

public class ReadSequentialFile extends JFrame {
    private ObjectInputStream input;
}
```

Fig. 16.8 Reading a sequential file (part 1 of 6).
private BankUI userInterface;
private JButton nextButton, openButton;

// Constructor -- initialize the Frame
public ReadSequentialFile()
{
    super( "Reading a Sequential File of Objects" );

    // create instance of reusable user interface
    userInterface = new BankUI( 4 ); // four textfields
    getContentPane().add(
        userInterface, BorderLayout.CENTER );

    // get reference to generic task button doTask1 from BankUI
    openButton = userInterface.getDoTask1Button();
    openButton.setText( "Open File" );

    // register listener to call openFile when button pressed
    openButton.addActionListener(
        new ActionListener() {

            // close file and terminate application
            public void actionPerformed( ActionEvent event )
            {
                openFile();

            } // end anonymous inner class

        } ); // end call to addActionListener

    // register window listener for window closing event
    addWindowListener(
        new WindowAdapter() {

            // close file and terminate application
            public void windowClosing( WindowEvent event )
            {
                if ( input != null )
                    closeFile();

                System.exit( 0 );

            } // end anonymous inner class

        } ); // end call to addWindowListener

Fig. 16.8 Reading a sequential file (part 2 of 6).
// get reference to generic task button doTask2 from BankUI
nextButton = userInterface.getDoTask2Button();
nextButton.setText( "Next Record" );
nextButton.setEnabled( false );

// register listener to call readRecord when button pressed
nextButton.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent event ) {
        readRecord();
    }
} );

pack();
setSize( 300, 200 );
show();

} // end ReadSequentialFile constructor

private void openFile() {
    // display file dialog so user can select file to open
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );

    int result = fileChooser.showOpenDialog( this );

    if ( result == JFileChooser.CANCEL_OPTION )
        return;

    // obtain selected file
    File fileName = fileChooser.getSelectedFile();

    // display error if file name invalid
    if ( fileName == null || fileName.getName().equals( "" ) )
        JOptionPane.showMessageDialog( this, "Invalid File Name", "Invalid File Name", JOptionPane.ERROR_MESSAGE );
    else {
        // display file dialog so user can select file to open
        JOptionPane.showMessageDialog( this, "Invalid File Name", "Invalid File Name", JOptionPane.ERROR_MESSAGE );
    }

}  // end anonymous inner class

}; // end call to addActionListener

// enable user to select file to open

Fig. 16.8 Reading a sequential file (part 3 of 6).
// open file
try {
    input = new ObjectInputStream(
        new FileInputStream( fileName ) );
    openButton.setEnabled( false );
    nextButton.setEnabled( true );
}

// process exceptions opening file
catch ( IOException ioException ) {
    JOptionPane.showMessageDialog( this,
        "Error Opening File", "Error",
        JOptionPane.ERROR_MESSAGE );
}

// end else
} // end method openFile

// read record from file
public void readRecord() {
    AccountRecord record;

    // input the values from the file
    try {
        record = ( AccountRecord ) input.readObject();

        // create array of Strings to display in GUI
        String values[] = {
            String.valueOf( record.getAccount() ),
            record.getFirstName(),
            record.getLastName(),
            String.valueOf( record.getBalance() )
        };

        // display record contents
        userInterface.setFieldValues( values );
    }

    // display message when end-of-file reached
    catch ( EOFException endOfFileException ) {
        nextButton.setEnabled( false );
        JOptionPane.showMessageDialog( this,
            "No more records in file",
            "End of File", JOptionPane.ERROR_MESSAGE );
    }

    // display error message if cannot read object
    // because class not found
    catch ( ClassNotFoundException classNotFoundException ) {
        JOptionPane.showMessageDialog( this,
            "Class Not Found", "Error",
            JOptionPane.ERROR_MESSAGE );
    }

    // display error message if cannot read object
    // because stream not found
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error Reading File", "Error",
            JOptionPane.ERROR_MESSAGE );
    }
}

Fig. 16.8 Reading a sequential file (part 4 of 6).
"Unable to create object",
"Class Not Found", JOptionPane.ERROR_MESSAGE );

    } // display error message if cannot read
    // due to problem with file
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
        "Error during read from file",
        "Read Error", JOptionPane.ERROR_MESSAGE );
    }

    } // close file and terminate application

private void closeFile()
{
    // close file and exit
    try {
        input.close();
        System.exit( 0 );
    }

    } // process exception while closing file
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
        "Error closing file",
        "Error", JOptionPane.ERROR_MESSAGE );

        System.exit( 1 );
    }

    } // execute application; ReadSequentialFile constructor
    // displays window
    public static void main( String args[] )
    {
        new ReadSequentialFile();
    }

} // end class ReadSequentialFile

Fig. 16.8 Reading a sequential file (part 5 of 6).
Most of the code in this example is similar to Fig. 16.6, so we discuss only the key lines of code that are different. Line 105 calls `JFileChooser` method `showOpenDialog` to display the Open dialog (second screen capture in Fig. 16.8). The behavior and GUI are the same as the dialog displayed by `showSaveDialog`, except that the title of the dialog and the Save button are both replaced with Open.

Lines 125–126 create a `ObjectInputStream` object and assign it to input. The `FileName` is passed to the `FileInputStream` constructor to open the file.

The program reads a record from the file each time the user clicks the Next Record button. Line 84 in Next Record’s actionPerformed method calls method `readRecord` (lines 144–187) to read one record from the file. Line 150 calls method `readObject` to read an Object from the `ObjectInputStream`. To use AccountRecord specific methods, we cast the returned Object to type AccountRecord. If the end-of-file marker is reached during reading, `readObject` throws an `EndOfFileException`.

To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data are found. It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program. Class `FileInputStream` does not provide the ability to reposition to the beginning of the file to read the file again unless the program closes the file and reopens it. Class `RandomAccessFile` objects can reposition to the beginning of the file. Class `RandomAccessFile` provides all the capabilities of the
classes FileInputStream, FileOutputStream, DataInputStream and DataOutputStream and adds several other methods, including a seek that repositions the file-position pointer (the byte number of the next byte in the file to be read or written) to any position in the file. However, class RandomAccessFile cannot read and write entire objects.

Performance Tip 16.5

The process of closing and reopening a file for the purpose of positioning the file-position pointer back to the beginning of a file is a time-consuming task for the computer. If this is done frequently, it can slow the performance of your program.

The program of Fig. 16.9 enables a credit manager to display the account information for those customers with zero balances (i.e., customers who do not owe the company any money), credit balances (i.e., customers to whom the company owes money) and debit balances (i.e., customers who owe the company money for goods and services received in the past).

The program displays buttons that allow a credit manager to obtain credit information. The Credit balances button produces a list of accounts with credit balances. The Debit balances button produces a list of accounts with debit balances. The Zero balances button produces a list of accounts with zero balances.

Records are displayed in a JTextArea called recordDisplayArea. The record information is collected by reading through the entire file and determining, for each record, whether it satisfies the criteria for the account type selected by the credit manager. Clicking one of the balance buttons sets variable accountType (line 274) to the clicked button’s text (e.g., Zero balances) and invokes method readRecords (191–238), which loops through the file and reads every record. Line 208 of method readRecords calls method shouldDisplay (241–259) to determine whether the current record satisfies the account type requested. If shouldDisplay returns true, the program appends the account information for the current record to the JTextArea recordDisplay. When the end-of-file marker is reached, line 219 calls method closeFile to close the file.

```java
// Fig. 16.9: CreditInquiry.java
// This program reads a file sequentially and displays the contents in a text area based on the type of account the user requests (credit balance, debit balance or zero balance).
// Java core packages
import java.io.*;
import java.awt.*;
import java.text.DecimalFormat;
// Java extension packages
import javax.swing.*;
// Deitel packages
import com.deitel.jhtp4.ch16.AccountRecord;
```

Fig. 16.9 Credit inquiry program (part 1 of 7).
public class CreditInquiry extends JFrame {
    private JTextArea recordDisplayArea;
    private JButton openButton,
        creditButton, debitButton, zeroButton;
    private JPanel buttonPanel;

    private ObjectInputStream input;
    private FileInputStream fileInput;
    private File fileName;
    private String accountType;

    // set up GUI
    public CreditInquiry()
    {
        super( "Credit Inquiry Program" );
        Container container = getContentPane();

        // set up panel for buttons
        buttonPanel = new JPanel();

        // create and configure button to open file
        openButton = new JButton( "Open File" );
        buttonPanel.add( openButton );

        // register openButton listener
        openButton.addActionListener(
            new ActionListener() {
                // anonymous inner class to handle openButton event
                public void actionPerformed( ActionEvent event )
                {
                    openFile( true );
                }
            } // end anonymous inner class
        ); // end call to addActionListener

        // create and configure button to get
        // accounts with credit balances
        creditButton = new JButton( "Credit balances" );
        buttonPanel.add( creditButton );
        creditButton.addActionListener( new ButtonHandler() );

        // create and configure button to get
        // accounts with debit balances
        debitButton = new JButton( "Debit balances" );
        buttonPanel.add( debitButton );
        debitButton.addActionListener( new ButtonHandler() );
    }

    // open file for processing
    public void actionPerformed( ActionEvent event )
    {
        openFile( true );
    }
}

Fig. 16.9 Credit inquiry program (part 2 of 7).
// create and configure button to get
// accounts with credit balances
zeroButton = new JButton( "Zero balances" );
buttonPanel.add( zeroButton );
zeroButton.addActionListener( new ButtonHandler() );

// set up display area
recordDisplayArea = new JTextArea();
JScrollPane scroller =
   new JScrollPane( recordDisplayArea );

// attach components to content pane
container.add( scroller, BorderLayout.CENTER );
container.add( buttonPanel, BorderLayout.SOUTH );

// disable creditButton, debitButton and zeroButton
creditButton.setEnabled( false );
debitButton.setEnabled( false );
zeroButton.setEnabled( false );

// register window listener
addWindowListener(
    new WindowAdapter() {

        // anonymous inner class for windowClosing event
        public void windowClosing( WindowEvent event )
        {
            closeFile();
            System.exit( 0 );
        }
    } // end anonymous inner class
); // end call to addWindowListener

// pack components and display window
pack();
setSize( 600, 250 );
show();

} // end CreditInquiry constructor

// enable user to choose file to open first time;
// otherwise, reopen chosen file
private void openFile( boolean firstTime )
{
    if ( firstTime ) {

        // display dialog, so user can choose file
        JFileChooser fileChooser = new JFileChooser();

fig. 16.9 Credit inquiry program (part 3 of 7).
fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);

int result = fileChooser.showOpenDialog(this);

// if user clicked Cancel button on dialog, return
if (result == JFileChooser.CANCEL_OPTION)
    return;

// obtain selected file
fileName = fileChooser.getSelectedFile();

// display error if file name invalid
if (fileName == null ||
    fileName.getName().equals(""))
    JOptionPane.showMessageDialog(this,
                                   "Invalid File Name",
                                   "Invalid File Name",
                                   JOptionPane.ERROR_MESSAGE);

else {

    // open file
    try {

        // close file from previous operation
        if (input != null)
            input.close();

        fileInput = new FileInputStream(fileName);
        input = new ObjectInputStream(fileInput);
        openButton.setEnabled(false);
        creditButton.setEnabled(true);
        debitButton.setEnabled(true);
        zeroButton.setEnabled(true);
    }

    // catch problems manipulating file
    catch (IOException ioException) {
        JOptionPane.showMessageDialog(this,
                                       "File does not exist",
                                       "Invalid File Name",
                                       JOptionPane.ERROR_MESSAGE);
    }

} // end method openFile

// close file before application terminates
private void closeFile()
{
    // close file
    try {
        input.close();
    }
}
// process exception from closing file
try {
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error closing file",
            "Error", JOptionPane.ERROR_MESSAGE );

        System.exit( 1 );
    }
}

// read records from file and display only records of
// appropriate type
private void readRecords()
{
    AccountRecord record;
    DecimalFormat twoDigits = new DecimalFormat( "0.00" );
    openFile( false );

    // read records
    try {
        recordDisplayArea.setText( "The accounts are:
        ");

        // input the values from the file
        while ( true ) {
            // read one AccountRecord
            record = ( AccountRecord ) input.readObject();

            // if proper account type, display record
            if ( shouldDisplay( record.getBalance() ) )
                recordDisplayArea.append( record.getAccount() +
                "\t" + record.getFirstName() + "\t" +
                record.getLastName() + "\t" +
                twoDigits.format( record.getBalance() ) +
                "\n" );
        }
    }

    // close file when end-of-file reached
    catch ( EOFException eofException ) {
        closeFile();
    }

    // display error if cannot read object
    // because class not found
    catch ( ClassNotFoundException classNotFound ) {
        JOptionPane.showMessageDialog( this,
            "Unable to create object",
            "Class Not Found", JOptionPane.ERROR_MESSAGE );
    }
}
// display error if cannot read
// because problem with file
try {
    // use file
    File file = new File( "credit.txt" );
    FileInputStream fis = new FileInputStream( file );
    DataInputStream dis = new DataInputStream( fis );
    // read data
    String accountType = dis.readUTF();
    double balance = dis.readDouble();
}
// end method readRecords

// uses record ty to determine if a record should be displayed
private boolean shouldDisplay( double balance )
{
    if ( accountType.equals( "Credit balances" ) && balance < 0 )
        return true;
    else if ( accountType.equals( "Debit balances" ) && balance > 0 )
        return true;
    else if ( accountType.equals( "Zero balances" ) && balance == 0 )
        return true;
    return false;
}

// execute application
public static void main( String args[] )
{
    new CreditInquiry();
}

// private inner class for creditButton, debitButton and zeroButton event handling
private class ButtonHandler implements ActionListener {
    // read records from file
    public void actionPerformed( ActionEvent event )
    {
        accountType = event.getActionCommand();
        readRecords();
    }
}

Fig. 16.9 Credit inquiry program (part 6 of 7).
16.6 Updating Sequential-Access Files

Data that is formatted and written to a sequential-access file as shown in Section 16.4 cannot be modified without reading and writing all the data in the file. For example, if the name White needed to be changed to Worthington, the old name cannot simply be overwritten. Such updating can be done, but it is awkward. To make the preceding name change, the records before White in a sequential-access file could be copied to a new file, the updated record would then be written to the new file, and the records after White would be copied to the new file. This requires processing every record in the file to update one record. If many records are being updated in one pass of the file, this technique can be acceptable.

Fig. 16.9 Credit inquiry program (part 7 of 7).


16.7 Random-Access Files

So far, we have seen how to create sequential-access files and to search through them to locate particular information. Sequential-access files are inappropriate for so-called “instant-access” applications, in which a particular record of information must be located immediately. Some popular instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated-teller machines and other kinds of transaction-processing systems that require rapid access to specific data. The bank at which you have your account might have hundreds of thousands or even millions of other customers, yet, when you use an automated teller machine, the bank determines in seconds whether your account has sufficient funds for the transaction. This kind of instant access is possible with random-access files. A program can access individual records of a random-access file directly (and quickly) without searching through other records. Random-access files are sometimes called direct-access files.

As we have said, Java does not impose structure on a file, so an application that wants to use random-access files must create them. Several techniques can be used to create random-access files. Perhaps the simplest is to require that all records in a file be of the same fixed length.

Using fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. We will soon see how this facilitates immediate access to specific records, even in large files.

Figure 16.10 illustrates Java’s view of a random-access file composed of fixed-length records (each record in this figure is 100 bytes long). A random-access file is like a railroad train with many cars—some empty, some with contents.

A program can insert data in a random-access file without destroying other data in the file. Also, a program can update or delete data stored previously without rewriting the entire file. In the following sections, we explain how to create a random-access file, enter data, read the data both sequentially and randomly, update the data and delete data no longer needed.

16.8 Creating a Random-Access File

RandomAccessFile objects have all the capabilities of DataInputStream and DataOutputStream objects discussed earlier. When a program associates an object of class RandomAccessFile with a file, the program reads or writes data beginning at the location in the file specified by the file-position pointer and manipulates all data as primitive data types. When writing an int value, 4 bytes are output to the file. When reading a double value, 8 bytes are input from the file. The size of the data types is guaranteed, because Java has fixed sizes for all primitive data types regardless of the computing platform.

Random-access file-processing programs rarely write a single field to a file. Normally, they write one object at a time, as we show in the following examples.

Consider the following problem statement:

Create a transaction-processing program capable of storing up to 100 fixed-length records for a company that can have up to 100 customers. Each record should consist of an account number that will be used as the record key, a last name, a first name and a balance. The program should be able to update an account, insert a new account and delete an account.
The next several sections introduce the techniques necessary to create this credit-processing program. Figure 16.11 contains the `RandomAccessAccountRecord` class that is used by the next four programs for both reading records from and writing records to a file.

```java
// Fig. 16.11: RandomAccessAccountRecord.java
// Subclass of AccountRecord for random access file programs.
package com.deitel.jhtp4.ch16;

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

public class RandomAccessAccountRecord extends AccountRecord {

    // no-argument constructor calls other constructor
    // with default values
    public RandomAccessAccountRecord() {
        this( 0, "", "", 0.0 );
    }

    // initialize a RandomAccessAccountRecord
    public RandomAccessAccountRecord( int account,
        String firstName, String lastName, double balance )
    {
        super( account, firstName, lastName, balance );
    }

    // read a record from specified RandomAccessFile
    public void read( RandomAccessFile file ) throws IOException {
        setAccount( file.readInt() );
        setFirstName( padName( file ) );
        setLastName( padName( file ) );
        setBalance( file.readDouble() );
    }
}
```

**Fig. 16.11** `RandomAccessAccountRecord` class used in the random-access file programs (part 1 of 2).
Class `RandomAccessAccountRecord` inherits `AccountRecord`'s (Fig. 16.5) implementation, which includes `private` instance variables—`account, lastName, firstName` and `balance`—as well as their `public` set and get methods.
Method `read` (lines 25–31) reads one record from the `RandomAccessFile` object passed as an argument. Methods `readInt` (line 27) and `readDouble` (line 30) read the `account` and `balance`, respectively. Method `read` calls `private` method `padName` (lines 34–45) twice to obtain the first and last names. Method `padName` reads fifteen characters from the `RandomAccessFile` and returns a `String`. If a name is shorter than 15 characters, the program fills each extra character with a null byte (`\0`). Swing components, such as `JTextField`s, cannot display null byte characters (which are displayed instead as rectangles). Line 44 solves this problem by replacing null bytes with spaces.

Method `write` (48–54) outputs one record to the `RandomAccessFile` object passed in as an argument. This method uses method `writeInt` to output the integer `account`, method `writeChars` (called from utility method `writeName`) to output the `firstName` and `lastName` character arrays and method `writeDouble` to output the double `balance`. [Note: In order to ensure that all records in the `RandomAccessFile` have the same size, we write exactly 15 characters for the first name and exactly 15 characters for the last name.] Method `writeName` (lines 57–69) performs the write operations for the first and last name.

Figure 16.12 illustrates opening a random-access file and writing data to the disk. This program writes 100 `RandomAccessAccountRecord`s, using method `write` (Fig. 16.11). Each `RandomAccessAccountRecord` object contains 0 for the account number, `null` for the last name, `null` for the first name and 0.0 for the balance. The file is initialized to create the proper amount of “empty” space in which the account data will be stored and to enable us to determine in subsequent programs whether each record is empty or contains data.

```java
// Fig. 16.12: CreateRandomFile.java
// This program creates a random access file sequentially
// by writing 100 empty records to disk.

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

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

// Deitel packages

public class CreateRandomFile {

    // enable user to select file to open
    private void createFile()
    {
        // display dialog so user can choose file
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
        int result = fileChooser.showSaveDialog(null);
```
// if user clicked Cancel button on dialog, return
if ( result == JFileChooser.CANCEL_OPTION )
    return;

// obtain selected file
File fileName = fileChooser.getSelectedFile();

// display error if file name invalid
if ( fileName == null ||
    fileName.getName().equals( "" ) )
    JOptionPane.showMessageDialog( null,
    "Invalid File Name", "Invalid File Name",
    JOptionPane.ERROR_MESSAGE );

else {
    // open file
    try {
        RandomAccessFile file =
            new RandomAccessFile( fileName, "rw" );
        RandomAccessAccountRecord blankRecord =
            new RandomAccessAccountRecord();

        // write 100 blank records
        for ( int count = 0; count < 100; count++ )
            blankRecord.write( file );

        // close file
        file.close();

        // display message that file was created
        JOptionPane.showMessageDialog( null,
        "Created file " + fileName, "Status",
        JOptionPane.INFORMATION_MESSAGE );

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

    // process exceptions during open, write or
    // close file operations
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( null,
        "Error processing file", "Error processing file",
        JOptionPane.ERROR_MESSAGE );

        System.exit( 1 );
    }
}  // end method openFile

Fig. 16.12 Creating a random-access file sequentially (part 2 of 3).
Lines 44–45 attempt to open a RandomAccessFile for use in this program. The RandomAccessFile constructor receives two arguments—the file name and the file open mode. The file open mode for a RandomAccessFile is either "r" to open the file for reading or "rw" to open the file for reading and writing.

If an IOException occurs during the open process, the program displays a message dialog and terminates. If the file opens properly, the program uses a for structure (lines 51–52) to invoke RandomAccessAccountRecord method write 100 times. This statement causes the data members of object blankRecord to be written to the file associated with RandomAccessFile object file.

### 16.9 Writing Data Randomly to a Random-Access File

Figure 16.13 writes data to a file that is opened with the "rw" mode for reading and writing. It uses the RandomAccessFile method seek to determine the exact location in the file at which a record of information is stored. Method seek sets the file-position pointer to a specific position in the file relative to the beginning of the file, and the RandomAccessAccountRecord class method write outputs the data. This program assumes
the user does not enter duplicate account numbers and that the user enters appropriate data in each JTextField.

```
public class WriteRandomFile extends JFrame {
    private RandomAccessFile output;
    private BankUI userInterface;
    private JButton enterButton, openButton;

    public WriteRandomFile() {
        super( "Write to random access file" );

        // create instance of reusable user interface BankUI
        userInterface = new BankUI( 4 ); // four textfields
        getContentPane().add( userInterface, BorderLayout.CENTER );

        // get reference to generic task button doTask1 in BankUI
        openButton = userInterface.getDoTask1Button();
        openButton.setText( "Open..." );

        // register listener to call openFile when button pressed
        openButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent event ) {
                    openFile();
                }
            }
        ); // end call to addActionListener
    }

    public void openFile() {
        // allow user to select file to open
        File file = JFileChooser.showOpenDialog( this );
        if ( file != null ) {
            try {
                output = new RandomAccessFile( file, "rw" );
            } catch ( FileNotFoundException e ) {
                JOptionPane.showMessageDialog( this, e.getMessage() );
            }
        }
    }
}
```

Fig. 16.13 Writing data randomly to a random-access file (part 1 of 5).
// register window listener for window closing event
addWindowListener(

    // anonymous inner class to handle windowClosing event
    new WindowAdapter() {

        // add record in GUI, then close file
        public void windowClosing( WindowEvent event )
        {
            if ( output != null )
                addRecord();

            closeFile();
        }
    }  // end anonymous inner class

); // end call to addWindowListener

// get reference to generic task button doTask2 in BankUI
enterButton = userInterface.getDoTask2Button();
enterButton.setText( "Enter" );
enterButton.set_enabled( false );

    // register listener to call addRecord when button pressed
    enterButton.addActionListener(

        // anonymous inner class to handle enterButton event
        new ActionListener() {

            // add record to file
            public void actionPerformed( ActionEvent event )
            {
                addRecord();
            }
        }  // end anonymous inner class

    ); // end call to addActionListener

setSize( 300, 150 );
show();

    // enable user to choose file to open
private void openFile()
{

    // display file dialog so user can select file
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );

int result = fileChooser.showOpenDialog( this );

// if user clicked Cancel button on dialog, return
if ( result == JFileChooser.CANCEL_OPTION )
    return;

// obtain selected file
File fileName = fileChooser.getSelectedFile();

// display error if file name invalid
if ( fileName == null ||
    fileName.getName().equals( "" ) )
    JOptionPane.showMessageDialog( this,
        "Invalid File Name", "Invalid File Name",
        JOptionPane.ERROR_MESSAGE );
else {

    // open file
    try {
        output = new RandomAccessFile( fileName, "rw" );
        enterButton.setEnabled( true );
        openButton.setEnabled( false );
    }

    // process exception while opening file
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "File does not exist",
            "Invalid File Name",
            JOptionPane.ERROR_MESSAGE );
    }
}
}  // end method openFile

// close file and terminate application
private void closeFile()
{
    // close file and exit
    try {
        if ( output != null )
            output.close();
    System.exit( 0 );
    }

    // process exception while closing file
    catch( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error closing file",
            "Error", JOptionPane.ERROR_MESSAGE );
    }
}

Fig. 16.13 Writing data randomly to a random-access file (part 3 of 5).
System.exit( 1 );

// add one record to file
public void addRecord()
{
    int accountNumber = 0;
    String fields[] = userInterface.getFieldValues();
    RandomAccessAccountRecord record =
        new RandomAccessAccountRecord();

    // ensure account field has a value
    if ( ! fields[ BankUI.ACCOUNT ].equals( "" ) ) {
        // output values to file
        try {
            accountNumber =
                Integer.parseInt( fields[ BankUI.ACCOUNT ] );

            if ( accountNumber > 0 && accountNumber <= 100 ) {
                record.setAccount( accountNumber );

                record.setFirstName( fields[ BankUI.FIRSTNAME ] );
                record.setLastName( fields[ BankUI.LASTNAME ] );
                record.setBalance( Double.parseDouble( fields[ BankUI.BALANCE ] ) );

                output.seek( ( accountNumber - 1 ) *
                    RandomAccessAccountRecord.size() );
                record.write( output );
            }

            userInterface.clearFields(); // clear TextFields
        }

    // process improper account number or balance format
    catch ( NumberFormatException formatException ) {
        JOptionPane.showMessageDialog( this,
            "Bad account number or balance",
            "Invalid Number Format",
            JOptionPane.ERROR_MESSAGE );
    }

    // process exceptions while writing to file
    catch ( IOException ioException ) {
        closeFile();
    }

} // end method addRecord

Fig. 16.13 Writing data randomly to a random-access file (part 4 of 5).
The user enters values for the account number, first name, last name and balance. When the user clicks the **Enter** button, the program calls method **addRecord** (162–207) of class **WriteRandomFile** to retrieve the data from the **BankAccountUI**’s JTextFields, store the data in **RandomAccessAccountRecord** class object **record** and call the **write** method of class **RandomAccessAccountRecord** to output the data.

Lines 185–186 call **RandomAccessFile** method **seek** to position the file-position pointer for object **output** to the byte location calculated by (accountNumber - 1) * **RandomAccessAccountRecord.size()**. Account numbers in this program should be between 1 and 100. We subtract 1 from the account number when calculating the byte location of the record. Thus, for record 1, the file-position pointer is set to byte 0 of the file.
When the user closes the window, the program attempts to add the last record to the file (if there is one in the GUI waiting to be output), closes the file and terminates.

16.10 Reading Data Sequentially from a Random-Access File

In the previous sections, we created a random-access file and wrote data to that file. In this section, we develop a program (Fig. 16.14) that opens a RandomAccessFile for reading with the "r" file open mode, reads through the file sequentially and displays only those records containing data. This program produces an additional benefit. See whether you can determine what it is; we will reveal it at the end of this section.

Good Programming Practice 16.1

Open a file with the "r" file open mode for input if the contents of the file should not be modified. This prevents unintentional modification of the file's contents. This is another example of the principle of least privilege.

```java
// Fig. 16.14: ReadRandomFile.java
// This program reads a random-access file sequentially and
// displays the contents one record at a time in text fields.

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.text.DecimalFormat;

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

// Deitel packages
import com.deitel.jhtp4.ch16.*;

public class ReadRandomFile extends JFrame {
    private BankUI userInterface;
    private RandomAccessFile input;
    private JButton nextButton, openButton;

    // set up GUI
    public ReadRandomFile()
    {
        super( "Read Client File" );

        // create reusable user interface instance
        userInterface = new BankUI( 4 ); // four textfields
        getContentPane().add( userInterface );

        // configure generic doTask1 button from BankUI
        openButton = userInterface.getDoTask1Button();
        openButton.setText( "Open File for Reading..." );
    }
}
```

Fig. 16.14 Reading a random-access file sequentially (part 1 of 5).
// register listener to call openFile when button pressed
openButton.addActionListener(
    // anonymous inner class to handle openButton event
    new ActionListener() {
        // enable user to select file to open
        public void actionPerformed( ActionEvent event ) {
            openFile();
        }
    } // end anonymous inner class
); // end call to addActionListener

// configure generic doTask2 button from BankUI
nextButton = userInterface.getDoTask2Button();
nextButton.setText( "Next" );
nextButton.setEnabled( false );

// register listener to call readRecord when button pressed
nextButton.addActionListener(
    // anonymous inner class to handle nextButton event
    new ActionListener() {
        // read a record when user clicks nextButton
        public void actionPerformed( ActionEvent event ) {
            readRecord();
        }
    } // end anonymous inner class
); // end call to addActionListener

// register listener for window closing event
addWindowListener(
    // anonymous inner class to handle windowClosing event
    new WindowAdapter() {
        // close file and terminate application
        public void windowClosing( WindowEvent event ) {
            closeFile();
        }
    } // end anonymous inner class
); // end call to addWindowListener

Fig. 16.14 Reading a random-access file sequentially (part 2 of 5).
```java
setSize(300, 150);
show();

// enable user to select file to open
private void openFile()
{
   // display file dialog so user can select file
   JFileChooser fileChooser = new JFileChooser();
   fileChooser.setFileSelectionMode(
      JFileChooser.FILES_ONLY);

   int result = fileChooser.showOpenDialog(this);

   // if user clicked Cancel button on dialog, return
   if (result == JFileChooser.CANCEL_OPTION)
      return;

   // obtain selected file
   File fileName = fileChooser.getSelectedFile();

   // display error is file name invalid
   if (fileName == null ||
      fileName.getName().equals("")
   )
      JOptionPane.showMessageDialog(this,
         "Invalid File Name", "Invalid File Name",
         JOptionPane.ERROR_MESSAGE);
   else {
      // open file
      try {
         input = new RandomAccessFile(fileName, "r");
         nextButton.setEnabled(true);
         openButton.setEnabled(false);
      } catch (IOException ioException) {
         JOptionPane.showMessageDialog(this,
            "File does not exist", "Invalid File Name",
            JOptionPane.ERROR_MESSAGE);
      }
   }
}

// read one record
public void readRecord()
{
   DecimalFormat twoDigits = new DecimalFormat("0.00");
   RandomAccessAccountRecord record =
      new RandomAccessAccountRecord();
```

---

Fig. 16.14 Reading a random-access file sequentially (part 3 of 5).
try {
  do {
    record.read( input );
  } while ( record.getAccount() == 0 );

  String values[] = {
    String.valueOf( record.getAccount() ),
    record.getFirstName(),
    record.getLastName(),
    String.valueOf( record.getBalance() )
  };
  userInterface.setFieldValues( values );
}

// close file when end-of-file reached
catch ( EOFException eofException ) {
  JOptionPane.showMessageDialog( this, "No more records",
    "End-of-file reached",
    JOptionPane.INFORMATION_MESSAGE);
  closeFile();
}

// process exceptions from problem with file
catch ( IOException ioException ) {
  JOptionPane.showMessageDialog( this, "Error Reading File",
    "Error",
    JOptionPane.ERROR_MESSAGE);
  System.exit( 1 );
}

} // end method readRecord

// close file and terminate application
private void closeFile()
{
  // close file and exit
  try {
    if ( input != null )
      input.close();
    System.exit( 0 );
  }

  // process exception closing file
  catch ( IOException ioException ) {
    JOptionPane.showMessageDialog( this, "Error closing file",
      "Error",
      JOptionPane.ERROR_MESSAGE );
    System.exit( 1 );
  }

Fig. 16.14 Reading a random-access file sequentially (part 4 of 5).
When the user clicks the Next button to read the next record in the file, the program invokes class ReadRandomFile's readRecord method (lines 136–174). This method invokes class RandomAccessAccountRecord's read method (line 146) to read the data into RandomAccessAccountRecord class object record. Method readRecord reads from the file until it encounters a record with a nonzero account number (zero is the initial value for the account). When readRecord encounters a valid account number (i.e., a nonzero value), the loop terminates, and readRecord displays the record data in the text fields. When the user clicks the Done button or when the end-of-file
marker is encountered while reading, method `closeFile` is invoked to close the file and terminate the program.

What about that additional benefit we promised? If you examine the GUI as the program executes, you will notice that the records are displayed in sorted order (by account number)! This is a simple consequence of the way we stored these records in the file, using direct-access techniques. Compared to the bubble sort we have seen (Chapter 7), sorting with direct-access techniques is blazingly fast. The speed is achieved by making the file large enough to hold every possible record that might be created, which enables the program to insert a record between other records without having to reorganize the file. This, of course, means that the file could be sparsely occupied most of the time, a waste of storage. So this is another example of the space/time trade-off. By using large amounts of space, we are able to develop a much faster sorting algorithm.

16.11 Example: A Transaction-Processing Program

We now present a substantial transaction-processing program (Fig. 16.20), using a random-access file to achieve “instant” access processing. The program maintains a bank’s account information. The program updates existing accounts, adds new accounts and deletes accounts. We assume that the program of Fig. 16.12 has been executed to create a file and that the program of Fig. 16.13 has been executed to insert initial data. The techniques used in this example have been presented in the earlier `RandomAccessFile` examples.

This program GUI consists of a window with a menu bar containing a `File` menu and internal frames that enable the user to perform insert, update and delete record operations on the file. The internal frames are subclasses of `JInternalFrame` that are managed by a `JDesktopPane` (as discussed in Chapter 13). The `File` menu has five menu items to select various tasks, as shown in Fig. 16.15.

When the user selects `Update Record` from the `File` menu, the `Update Record` internal frame (Fig. 16.16) allows the user to update an existing account. The code that implements the `Update Record` internal frame is in class `UpdateDialog` (lines 238–458 of Fig. 16.20). In the first screen capture of Fig. 16.16, the user inputs an account number and presses `Enter` to invoke the account textfield’s `actionPerformed` method (lines 345–360 of Fig. 16.20). This reads the account from the file with method `getRecord` (lines 371–418 of Fig. 16.20), which validates the account number, then reads the record with `RandomAccessAccountRecord` method `read`. Next, `getRecord` compares the account number with zero (i.e., no record) to determine whether the record contains information. If not, `getRecord` displays a message stating that the record does not exist; otherwise, it returns the record. Lines 350–357 of the account textfield’s `actionPerformed` method extract the account information from the record and display the account information in the internal frame (as shown in the second screen capture of Fig. 16.16). The `Transaction amount` textfield initially contains the string `charge (+)` or `payment (–)`. The user should select this text and type the transaction amount (a positive value for a charge or a negative value for a payment), then press `Enter` to invoke the transaction textfield’s `actionPerformed` method (lines 295–330 of Fig. 16.20). Method `addRecord` (lines 421–456 of Fig. 16.20) takes the transaction amount, adds it to the current balance and calls `RandomAccessAccountRecord` method `setBalance` to update the display. Clicking `Save Changes` writes the updated record to disk; clicking `Cancel` closes the internal frame without writing the record to disk. The windows in Fig. 16.17 show a sample of a transaction being input.
When the user selects **New Record** from the **File** menu, the **New Record** internal frame in Fig. 16.18 allows the user to add a new record. The code that implements the **New Record** internal frame is in class **NewDialog** (lines 461–615 of Fig. 16.20). The user enters data in the **JTextField**s and clicks **Save Changes** to write the record to disk. If the account number already exists, the program displays an error message and does not attempt to write the record. Clicking **Cancel** closes the internal frame without attempting to write the record.

Selecting **Delete Record** from the **File** menu displays the **Delete Record** internal frame in Fig. 16.19, which allows the user to delete a record from the file. The code that implements the **Delete Record** internal frame is in class **DeleteDialog** (lines 618–774 of Fig. 16.20). The user enters the account number in the **JTextField** and presses **Enter**. Only an existing record can be deleted, so, if the specified account is empty, the program displays an error message. This enables the user to check whether the record exists before deleting the record. Clicking the **Delete Record** button in the internal frame sets the record’s account number to 0 (which this application considers to be an empty record). Clicking **Cancel** closes the internal frame without deleting the record.
Type transaction amount and press the *Enter* key to update balance in dialog.

Press *Save Changes* to store new balance in file.

**Fig. 16.17** Inputting a transaction in the *Update Record* internal frame.

**Fig. 16.18** *New Record* internal frame.

**Fig. 16.19** *Delete Record* internal frame.
The TransactionProcessor program appears in Fig. 16.20. The program opens the file with file open mode "rw" (reading and writing).

```java
// Transaction processing program using RandomAccessFiles.
// This program reads a random-access file sequentially,
// updates records already written to the file, creates new
// records to be placed in the file and deletes data
// already in the file.

// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.text.DecimalFormat;

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

// Deitel packages
import com.deitel.jhtp4.ch16. *;

public class TransactionProcessor extends JFrame {
    private UpdateDialog updateDialog;
    private NewDialog newDialog;
    private DeleteDialog deleteDialog;
    private JMenuItem newItem, updateItem, deleteItem,
    openItem, exitItem;
    private JDesktopPane desktop;
    private RandomAccessFile file;
    private RandomAccessAccountRecord record;

    // set up GUI
    public TransactionProcessor()
    {
        super( "Transaction Processor" );

        // set up desktop, menu bar and File menu
desktop = new JDesktopPane();
getContentPane().add( desktop );

        JMenuBar menuBar = new JMenuBar();
setJMenuBar( menuBar );

        JMenu fileMenu = new JMenu( "File" );
menuBar.add( fileMenu );

        // set up menu item for adding a record
    newItem = new JMenuItem( "New Record" );
    newItem.setEnabled( false );
    newItem.setEnable( false );
```

Fig. 16.20 Transaction-processing program (part 1 of 15).
// display new record dialog when user selects New Record
newItem.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            newDialog.setVisible( true );
        }
    });

// set up menu item for updating a record
updateItem = new JMenuItem( "Update Record" );
updateItem.setEnabled( false );

// display update dialog when user selects Update Record
updateItem.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            updateDialog.setVisible( true );
        }
    });

// set up menu item for deleting a record
deleteItem = new JMenuItem( "Delete Record" );
deleteItem.setEnabled( false );

// display delete dialog when user selects Delete Record
deleteItem.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            deleteDialog.setVisible( true );
        }
    });

// set up button for opening file
openItem = new JMenuItem( "New/Open File" );

// enable user to select file to open, then set up
// dialog boxes
openItem.addActionListener(
    new ActionListener() {
        
});

Fig. 16.20 Transaction-processing program (part 2 of 15).
public void actionPerformed( ActionEvent event )
{
    boolean opened = openFile();

    if ( !opened )
        return;

    openItem.setEnabled( false );

    // set up internal frames for record processing
    updateDialog = new UpdateDialog( file );
    desktop.add( updateDialog );

    deleteDialog = new DeleteDialog( file );
    desktop.add( deleteDialog );

    newDialog = new NewDialog( file );
    desktop.add( newDialog );
}

} // end anonymous inner class

// set up menu item for exiting program
exitItem = new JMenuItem( "Exit" );
exitItem.setEnabled( true );

// terminate application
exitItem.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent event )
    {
        closeFile();
    }
};

// attach menu items to File menu
fileMenu.add( openItem );
fileMenu.add( newItem );
fileMenu.add( updateItem );
fileMenu.add( deleteItem );
fileMenu.addSeparator();
fileMenu.add( exitItem );

// configure window
setDefaultCloseOperation( WindowConstants.DO NOTHING ON CLOSE );
setSize( 400, 250 );

Fig. 16.20 Transaction-processing program (part 3 of 15).
setVisible( true );
}

// enable user to select file to open
private boolean openFile()
{
    // display dialog so user can select file
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
    int result = fileChooser.showOpenDialog( this );
    // if user clicked Cancel button on dialog, return
    if ( result == JFileChooser.CANCEL_OPTION )
        return false;
    // obtain selected file
    File fileName = fileChooser.getSelectedFile();
    // display error if file name invalid
    if ( fileName == null || fileName.getName().equals( "" ) ) {
        JOptionPane.showMessageDialog( this, "Invalid File Name", "Invalid File Name",
                                   JOptionPane.ERROR_MESSAGE );
        return false;
    }
    else {
        // open file
        try {
            file = new RandomAccessFile( fileName, "rw" );
            openItem.setEnabled( false );
            newItem.setEnabled( true );
            updateItem.setEnabled( true );
            deleteItem.setEnabled( true );
        }
        // process problems opening file
        catch ( IOException ioException ) {
            JOptionPane.showMessageDialog( this, "File does not exist", "Invalid File Name",
                                         JOptionPane.ERROR_MESSAGE );
            return false;
        }
    }
    return true; // file opened

Fig. 16.20  Transaction-processing program (part 4 of 15).
// close file and terminate application
private void closeFile()
{
    // close file and exit
    try {
        if ( file != null )
        file.close();
        System.exit( 0 );
    }

    // process exceptions closing file
    catch( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
        "Error closing file",
        "Error", JOptionPane.ERROR_MESSAGE );
        System.exit( 1 );
    }

    // execute application
    public static void main( String args[] )
    {
        new TransactionProcessor();
    }
}

// class for updating records
class UpdateDialog extends JInternalFrame {
    private RandomAccessFile file;
    private BankUI userInterface;

    // set up GUI
    public UpdateDialog( RandomAccessFile updateFile )
    {
        super( "Update Record" );
        file = updateFile;

        // set up GUI components
        userInterface = new BankUI( 5 );
        getContentPane().add( userInterface,
        BorderLayout.CENTER );

        // set up Save Changes button and register listener
        JButton saveButton = userInterface.getDoTask1Button();
        saveButton.setText( "Save Changes" );
        saveButton.addActionListener( Fig. 16.20 Transaction-processing program (part 5 of 15).
new ActionListener() {
  
  public void actionPerformed((ActionEvent event )
  {
    addRecord( getRecord() );
    setVisible( false );
    userInterface.clearFields();
  }
  }
};

// set up Cancel button and register listener
JButton cancelButton = userInterface.getDoTask2Button();
cancelButton.setText( "Cancel" );
cancelButton.addActionListener( 
  new ActionListener() {
    public void actionPerformed((ActionEvent event )
    {
      setVisible( false );
      userInterface.clearFields();
    }
  } );

// set up listener for transaction textfield
JTextField transactionField =
  userInterface.getFields()[ BankUI.TRANSACTION ];
transactionField.addActionListener( 
  new ActionListener() {
    public void actionPerformed((ActionEvent event )
    {
      // add transaction amount to balance
      try {
        RandomAccessAccountRecord record = getRecord();
        // get textfield values from userInterface
        String fieldValues[] =
          userInterface.getFieldValues();
        // get transaction amount
        double change = Double.parseDouble( 
          fieldValues[ BankUI.TRANSACTION ] );
        // specify Strings to display in GUI
        String[] values = {
          String.valueOf( record.getAccount() ),
          record.getFirstName(),
        }
      } catch (ParseException e) {
        // catch exception
      }
    }
  });
record.getLastName(),
    String.valueOf( record.getBalance()
    + change ),
    "Charge(+) or payment (-)" 
};

// display Strings in GUI
userInterface.setFieldValues( values );
}

// process invalid number in transaction field
try ( NumberFormatException numberFormat ) {
    JOptionPane.showMessageDialog( null,
        "Invalid Transaction",
        "Invalid Number Format",
        JOptionPane.ERROR_MESSAGE );
}

}  // end method actionPerformed

}  // end anonymous inner class

}); // end call to addActionListener

// set up listener for account text field
JTextField accountField =
    userInterface.getFields()[ BankUI.ACCOUNT ];
accountField.addActionListener(
    new ActionListener() {

        // get record and display contents in GUI
        public void actionPerformed( ActionEvent event )
        {
            RandomAccessAccountRecord record = getRecord();

            if ( record.getAccount() != 0 )
            {
                String values[] = {
                    String.valueOf( record.getAccount() ),
                    record.getFirstName(),
                    record.getLastName(),
                    String.valueOf( record.getBalance() ),
                    "Charge(+) or payment (-)" 
                };

                userInterface.setFieldValues( values );
            }
        }

    } // end method actionPerformed

}  // end anonymous inner class

}); // end call to addActionListener

Fig. 16.20 Transaction-processing program (part 7 of 15).
setSize( 300, 175 );
setSize( false );

// get record from file
private RandomAccessAccountRecord getRecord()
{
    RandomAccessAccountRecord record =
        new RandomAccessAccountRecord();

    // get record from file
    try {
        JTextField accountField =
            userInterface.getFields()[ BankUI.ACCOUNT ];

        int accountNumber =
            Integer.parseInt( accountField.getText() );

        if ( accountNumber < 1 || accountNumber > 100 ) {
            JOptionPane.showMessageDialog( this,
                "Account Does Not Exist",
                "Error", JOptionPane.ERROR_MESSAGE );
            return record;
        }

        // seek to appropriate record location in file
        file.seek( ( accountNumber - 1 ) *
            RandomAccessAccountRecord.size() );
        record.read( file );

        if ( record.getAccount() == 0 )
            JOptionPane.showMessageDialog( this,
                "Account Does Not Exist",
                "Error", JOptionPane.ERROR_MESSAGE );
    }

    // process invalid account number format
    catch ( NumberFormatException numberFormatException ) {
        JOptionPane.showMessageDialog( this,
            "Invalid Account", "Invalid Number Format",
            JOptionPane.ERROR_MESSAGE );
    }

    // process file processing problems
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error Reading File",
            "Error", JOptionPane.ERROR_MESSAGE );
    }

    return record;
}

Fig. 16.20 Transaction-processing program (part 8 of 15).
// add record to file
public void addRecord( RandomAccessAccountRecord record )
{
    // update record in file
    try {
        int accountNumber = record.getAccount();

        file.seek( ( accountNumber - 1 ) *
                   RandomAccessAccountRecord.size() );

        String[] values = userInterface.getFieldValues();

        // set firstName, lastName and balance in record
        record.setFirstName( values[ BankUI.FIRSTNAME ] );
        record.setLastName( values[ BankUI.LASTNAME ] );
        record.setBalance( Double.parseDouble( values[ BankUI.BALANCE ] ) );

        // rewrite record to file
        record.write( file );
    }
}

    // process file processing problems
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
                                        "Error Writing To File",
                                        "Error", JOptionPane.ERROR_MESSAGE );
    }

    // process invalid balance value
    catch ( NumberFormatException numberFormat ) {
        JOptionPane.showMessageDialog( this,
                                        "Bad Balance", "Invalid Number Format",
                                        JOptionPane.ERROR_MESSAGE );
    }

} // end method addRecord

} // end class UpdateDialog

// class for creating new records
class NewDialog extends JInternalFrame {
    private RandomAccessFile file;
    private BankUI userInterface;

    // set up GUI
    public NewDialog( RandomAccessFile newFile )
    {
        super( "New Record" );

        file = newFile;
    }

Fig. 16.20 Transaction-processing program (part 9 of 15).
// attach user interface to dialog
userInterface = new BankUI( 4 );
getContentPane().add( userInterface,
        BorderLayout.CENTER );

// set up Save Changes button and register listener
JButton saveButton = userInterface.getDoTask1Button();
saveButton.setText( "Save Changes" );

saveButton.addActionListener(
        new ActionListener() {
        // add new record to file
        public void actionPerformed( ActionEvent event )
        {
            addRecord( getRecord() );
            setVisible( false );
            userInterface.clearFields();
        }
    } // end anonymous inner class
); // end call to addActionListener

JButton cancelButton = userInterface.getDoTask2Button();
cancelButton.setText( "Cancel" );
cancelButton.addActionListener(
        new ActionListener() {
        // dismiss dialog without storing new record
        public void actionPerformed( ActionEvent event )
        {
            setVisible( false );
            userInterface.clearFields();
        }
    } // end anonymous inner class
); // end call to addActionListener

setSize( 300, 150 );
setVisible( false );

} // end constructor

// get record from file
private RandomAccessAccountRecord getRecord()
{
    RandomAccessAccountRecord record =
        new RandomAccessAccountRecord();

    // get record from file
    String recordString =
        // code to read record from file
        // ...
        // code to parse recordString
        account =
            new Account( recordString );

    return account;
}
// get record from file
try {
    JTextField accountField =
        userInterface.getFields()[ BankUI.ACCOUNT ];
    int accountNumber =
        Integer.parseInt( accountField.getText() );
    if ( accountNumber < 1 || accountNumber > 100 ) {
        JOptionPane.showMessageDialog( this,
            "Account Does Not Exist",
            "Error", JOptionPane.ERROR_MESSAGE );
        return record;
    }
    // seek to record location
    file.seek( ( accountNumber - 1 ) *
        RandomAccessAccountRecord.size() );
    // read record from file
    record.read( file );
}
// process invalid account number format
catch ( NumberFormatException numberFormat ) {
    JOptionPane.showMessageDialog( this,
        "Account Does Not Exist", "Invalid Number Format",
        JOptionPane.ERROR_MESSAGE );
}
// process file processing problems
catch ( IOException ioException ) {
    JOptionPane.showMessageDialog( this,
        "Error Reading File",
        "Error", JOptionPane.ERROR_MESSAGE );
}
return record;
}  // end method getRecord

// add record to file
public void addRecord( RandomAccessAccountRecord record )
{
    String[] fields = userInterface.getFieldValues();
    if ( record.getAccount() != 0 ) {
        JOptionPane.showMessageDialog( this,
            "Record Already Exists",
            "Error", JOptionPane.ERROR_MESSAGE );
        return;
    }
}  // end method addRecord
// output the values to the file
try {

    // set account, first name, last name and balance
    // for record
    record.setAccount( Integer.parseInt( fields[ BankUI.ACCOUNT ] ) );
    record.setFirstName( fields[ BankUI.FIRSTNAME ] );
    record.setLastName( fields[ BankUI.LASTNAME ] );
    record.setBalance( Double.parseDouble( fields[ BankUI.BALANCE ] ) );

    // seek to record location
    file.seek( ( record.getAccount() - 1 ) *
        RandomAccessAccountRecord.size() );

    // write record
    record.write( file );
}

    // process invalid account or balance format
    catch ( NumberFormatException numberFormat ) {
        JOptionPane.showMessageDialog( this,
            "Invalid Balance", "Invalid Number Format",
            JOptionPane.ERROR_MESSAGE );
    }

    // process file processing problems
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error Writing To File",
            "Error", JOptionPane.ERROR_MESSAGE );
    }
} // end method addRecord

} // end class NewDialog

// class for deleting records
class DeleteDialog extends JInternalFrame {
    private RandomAccessFile file;  // file for output
    private BankUI userInterface;

    // set up GUI
    public DeleteDialog( RandomAccessFile deleteFile )
    {
        super( "Delete Record" );

        file = deleteFile;

        // create BankUI with only account field
        userInterface = new BankUI( 1 );

getContentPane().add( userInterface,
    BorderLayout.CENTER );

    // set up Delete Record button and register listener
    JButton deleteButton = userInterface.getDoTask1Button();
    deleteButton.setText( "Delete Record" );
    deleteButton.addActionListener(
        new ActionListener() {
            // overwrite existing record
            public void actionPerformed( ActionEvent event )
            {
                addRecord( getRecord() );
                setVisible( false );
                userInterface.clearFields();
            }
        }   // end anonymous inner class
    ); // end call to addActionListener

    // set up Cancel button and register listener
    JButton cancelButton = userInterface.getDoTask2Button();
    cancelButton.setText( "Cancel" );
    cancelButton.addActionListener(
        new ActionListener() {
            // cancel delete operation by hiding dialog
            public void actionPerformed( ActionEvent event )
            {
                setVisible( false );
            }
        }   // end anonymous inner class
    ); // end call to addActionListener

    // set up listener for account text field
    JTextField accountField =
        userInterface.getFields()[ BankUI.ACCOUNT ];
    accountField.addActionListener(
        new ActionListener() {
            public void actionPerformed( ActionEvent event )
            {
                RandomAccessAccountRecord record = getRecord();
getRecord()
{
    RandomAccessAccountRecord record =
        new RandomAccessAccountRecord();
    // get record from file
    try {
        JTextField accountField =
            userInterface.getFields()[BankUI.ACCOUNT];
        int accountNumber =
            Integer.parseInt( accountField.getText() );

        if ( accountNumber < 1 || accountNumber > 100 ) {
            JOptionPane.showMessageDialog( this,
                "Account Does Not Exist",
                "Error", JOptionPane.ERROR_MESSAGE );
            return( record );
        }

        // seek to record location and read record
        file.seek( ( accountNumber - 1 ) *
            RandomAccessAccountRecord.size() );
        record.read( file );

        if ( record.getAccount() == 0 )
            JOptionPane.showMessageDialog( this,
                "Account Does Not Exist",
                "Error", JOptionPane.ERROR_MESSAGE );
    }
    // process invalid account number format
    catch ( NumberFormatException numberFormat ) {
        JOptionPane.showMessageDialog( this,
            "Account Does Not Exist",
            "Invalid Number Format",
            JOptionPane.ERROR_MESSAGE );
    }
    // process file processing problems
    catch ( IOException ioException ) {

As we stated at the beginning of this chapter, the java.io package contains an abundance of classes for processing input and output. We concentrated on the classes for processing sequential files (FileInputStream and FileOutputStream), for processing object streams (ObjectInputStream and ObjectOutputStream) and for processing random-access files (RandomAccessFile). In this section, we discuss class File, which is particularly useful for retrieving information about a file or a directory from a disk. Objects of class File do not open files or provide any file-processing capabilities.

One application of a File object is to determine whether a file exists before attempting to open the file. In Common Programming Error 16.1, we warned that opening an existing file for output by using a FileOutputStream object discards the contents

```java
JOptionPane.showMessageDialog( this,
    "Error Reading File",
    "Error", JOptionPane.ERROR_MESSAGE );
}
return record;
} // end method getRecord

// add record to file
public void addRecord( RandomAccessAccountRecord record )
{
    if ( record.getAccount() == 0 )
        return;
    // delete record by setting account number to 0
    try {
        int accountNumber = record.getAccount();
        // seek to record position
        file.seek( ( accountNumber - 1 ) *
            RandomAccessAccountRecord.size() );
        // set account to 0 and overwrite record
        record.setAccount( 0 );
        record.write( file );
    }
    // process file processing problems
    catch ( IOException ioException ) {
        JOptionPane.showMessageDialog( this,
            "Error Writing To File",
            "Error", JOptionPane.ERROR_MESSAGE );
    }
} // end method addRecord

} // end class DeleteDialog

Fig. 16.20 Transaction-processing program (part 15 of 15).

16.12 Class File
of that file without warning. If a program using a `File` determines that a file already exists, the program can warn that the user is about to discard the original file’s contents.

**Good Programming Practice 16.2**

Use a `File` object to determine whether a file exists before opening the file with a `FileOutputStream` object.

Class `File` provides three constructors. The constructor

```
public File( String name )
```

stores the `String` argument `name` in the object. The `name` can contain path information as well as a file or directory name. A file or directory’s path specifies the location of the file or directory on disk. The path includes some or all of the directories leading to the file or directory. An absolute path contains all the directories, starting with the root directory that lead to a specific file or directory. Every file or directory on a particular disk drive has the same root directory in its path. A relative path contains a subset of the directories leading to a specific file or directory. Relative paths normally start from the directory in which the application began executing.

The constructor

```
public File( String pathToName, String name )
```

uses argument `pathToName` (an absolute or relative path) to locate the file or directory specified by `name`.

The constructor

```
public File( File directory, String name )
```

uses an existing `File` object `directory` (an absolute or relative path) to locate the file or directory specified by `name`.

Figure 16.21 discusses some common `File` methods. See the Java API for other `File` methods.

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean canRead()</td>
<td>Returns true if a file is readable; false otherwise.</td>
</tr>
<tr>
<td>boolean canWrite()</td>
<td>Returns true if a file is writable; false otherwise.</td>
</tr>
<tr>
<td>boolean exists()</td>
<td>Returns true if the name specified as the argument to the <code>File</code> constructor is a file or directory in the specified path; false otherwise.</td>
</tr>
<tr>
<td>boolean isFile()</td>
<td>Returns true if the name specified as the argument to the <code>File</code> constructor is a file; false otherwise.</td>
</tr>
<tr>
<td>boolean isDirectory()</td>
<td>Returns true if the name specified as the argument to the <code>File</code> constructor is a directory; false otherwise.</td>
</tr>
<tr>
<td>boolean isAbsolute()</td>
<td>Returns true if the arguments specified to the <code>File</code> constructor indicate an absolute path to a file or directory; false otherwise.</td>
</tr>
</tbody>
</table>
```

*Fig. 16.21* Some commonly used `File` methods (part 1 of 2).
### Good Programming Practice 16.3

Use `File` method `isFile` to determine that a `File` object represents a file (not a directory) before attempting to open a file.

### Good Programming Practice 16.4

Before attempting to open a file for reading, use `File` method `canRead` to determine whether the file is readable.

### Good Programming Practice 16.5

Before attempting to open a file for writing, use `File` method `canWrite` to determine whether the file is writable.

Figure 16.22 demonstrates class `File`. The `FileTest` application creates a GUI containing a `JTextField` for entering a file name or directory name and a `JTextArea` for displaying information about the file name or directory name input.

```java
// Fig. 16.22: FileTest.java
// Demonstrating the File class.
// Java core packages
import java.awt.*;
import java.awt.event.*;
import java.io.*;
```

---

**Method** | **Description**
--- | ---
`String getAbsolutePath()` | Returns a `String` with the absolute path of the file or directory.
`String getName()` | Returns a `String` with the name of the file or directory.
`String getPath()` | Returns a `String` with the path of the file or directory.
`String getParent()` | Returns a `String` with the parent directory of the file or directory—that is, the directory in which the file or directory can be found.
`long length()` | Returns the length of the file in bytes. If the `File` object represents a directory, 0 is returned.
`long lastModified()` | Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is only useful for comparison with other values returned by this method.
`String[] list()` | Returns an array of `String`s representing the contents of a directory.

---

**Fig. 16.21** Some commonly used `File` methods (part 2 of 2).
public class FileTest extends JFrame implements ActionListener {

private JTextField enterField;
private JTextArea outputArea;

// set up GUI
public FileTest()
{
  super( "Testing class File" );

  enterField = new JTextField(
    "Enter file or directory name here" );
  enterField.addActionListener( this );
  outputArea = new JTextArea();

  ScrollPane scrollPane = new ScrollPane();
  scrollPane.add( outputArea );

  Container container = getContentPane();
  container.add( enterField, BorderLayout.NORTH );
  container.add( scrollPane, BorderLayout.CENTER );

  setSize( 400, 400 );
  show();
}

// display information about file user specifies
public void actionPerformed( ActionEvent actionEvent )
{
  File name = new File( actionEvent.getActionCommand() );

  // if name exists, output information about it
  if ( name.exists() ) {
    outputArea.setText(
      name.getName() + " exists
      ( name.isFile() ? "is a file\n      ( name.isDirectory() ? "is a directory\n      ( name.isAbsolute() ? "is absolute path\n      "Last modified: " + name.lastModified() +
      "\nLength: " + name.length() +
      "\nPath: " + name.getPath() +
      "\nAbsolute path: " + name.getAbsolutePath() +
      "\nParent: " + name.getParent() );

    // output information if name is a file
    if ( name.isFile() ) {
      // display information about file user specifies
      public void actionPerformed( ActionEvent actionEvent )
      {
        File name = new File( actionEvent.getActionCommand() );

        // if name exists, output information about it
        if ( name.exists() ) {
          outputArea.setText(
            name.getName() + " exists
            ( name.isFile() ? "is a file\n            ( name.isDirectory() ? "is a directory\n            ( name.isAbsolute() ? "is absolute path\n            "Last modified: " + name.lastModified() +
            "\nLength: " + name.length() +
            "\nPath: " + name.getPath() +
            "\nAbsolute path: " + name.getAbsolutePath() +
            "\nParent: " + name.getParent() );

          // output information if name is a file
          if ( name.isFile() ) {

Fig. 16.22 Demonstrating class File (part 2 of 4).
// append contents of file to outputArea
try {
    BufferedReader input = new BufferedReader( 
        new FileReader( name ) );
    StringBuffer buffer = new StringBuffer();
    String text;
    outputArea.append( "\n\n" );

    while ( ( text = input.readLine() ) != null )
        buffer.append( text + "\n" );

    outputArea.append( buffer.toString() );
}

// process file processing problems
catch( IOException ioException ) {
    JOptionPane.showMessageDialog( this, 
    "FILE ERROR",
    "FILE ERROR", JOptionPane.ERROR_MESSAGE );
}

// output directory listing
else if ( name.isDirectory() ) {
    String directory[] = name.list();

    outputArea.append( "\n\nDirectory contents:\n");

    for ( int i = 0; i < directory.length; i++ )
        outputArea.append( directory[ i ] + "\n" );
}

// not file or directory, output error message
else {
    JOptionPane.showMessageDialog( this, 
    actionEvent.getActionCommand() + " Does Not Exist",
    "ERROR", JOptionPane.ERROR_MESSAGE );
}

} // end method actionPerformed

// execute application
public static void main( String args[] )
{
    FileTest application = new FileTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

} // end class FileTest

Fig. 16.22 Demonstrating class File (part 3 of 4).
The user types a file name or directory name into the text field and presses the Enter key to invoke method `actionPerformed` (lines 40–103), which creates a new `File` object (line 42) and assigns it to `name`. Line 45 invokes `File` method `exists` to determine whether the name input by the user exists (either as a file or as a directory) on the disk. If the name input by the user does not exist, the `actionPerformed` method proceeds to lines 97–101 and displays a message dialog containing the name the user typed, followed by “DoesNotExist.” Otherwise, the body of the `if` structure (lines 45–94) executes. The program outputs the name of the file or directory, then outputs the results of testing the `File` object with `isFile` (line 48), `isDirectory` (line 50) and `isAbsolute` (line 52). Next, the program displays the values returned by `lastModified` (line 54),
length (line 55), getPath (line 56), getAbsolutePath (line 57) and getParent (line 58).

If the File object represents a file (line 61), the program reads the contents of the file and displays the contents in the JTextArea. The program uses a BufferedReader object chained to a FileReader object (lines 65–66) to open the file for reading and to read the file one line at a time with method readLine (line 71). Note that the FileReader object was initialized with the File object name (line 66).

If the File object represents a directory (line 86), the program reads the contents of the directory into the program by using File method list, then displays the directory contents in the JTextArea.

The first output demonstrates a File object associated with the jfc directory from the Java 2 Software Development Kit. The second output of this program demonstrates a File object associated with the readme.txt file from the Java 2 Software Development Kit. In both cases, we specified an absolute path on our personal computer.

Note that the \ separator character is used to separate directories and files in the path. On a UNIX workstation, the separator character would be the / character. Java processes both characters identically in a path name. So, if we specified the path

```
c:\java/readme.txt
```

which uses one of each separator character, Java still processes the file properly.

Common Programming Error 16.2

Using \ as a directory separator rather than \ in a string literal is a logic error. A single \\ indicates that the \ and the next character represent an escape sequence. To insert a \ in a string literal, you must use \\.

Good Programming Practice 16.6

When building Strings that represent path information, use File.separatorChar to obtain the local computer’s proper separator character, rather than explicitly using / or \.

SUMMARY

• All data items processed by a computer are reduced to combinations of zeros and ones.
• The smallest data item in a computer (a bit) can assume the value 0 or the value 1.
• Digits, letters and special symbols are called characters. The set of all characters used to write programs and represent data items on a particular computer is called that computer’s character set. Every character in a computer’s character set is represented as a pattern of 1s and 0s. (Characters in Java are Unicode characters composed of 2 bytes.)
• A field is a group of characters (or bytes) that conveys meaning.
• A record is a group of related fields.
• At least one field in a record is chosen as a record key to identify a record as belonging to a particular person or entity that is unique from all other records in the file.
• Java imposes no structure on a file. Notions like “record” do not exist in Java. The programmer must structure a file appropriately to meet the requirements of an application.
• A collection of programs designed to create and manage databases is called a database management system (DBMS).
• Java views each file as a sequential stream of bytes.
• Each file ends in some machine-dependent form of end-of-file marker.

• Streams provide communication channels between programs and files, memory or other programs across a network.

• Programs use classes from package java.io to perform Java file I/O. This package includes the definitions for the stream classes, such as FileInputStream, FileOutputStream, DataInputStream and DataOutputStream.

• Files are opened by instantiating objects of stream classes FileInputStream, FileOutputStream, RandomAccessFile, FileReader and FileWriter.

• InputStream (a subclass of Object) and OutputStream (a subclass of Object) are abstract classes that define methods for performing input and output, respectively.

• File input/output is done with FileInputStream (a subclass of InputStream) and FileOutputStream (a subclass of OutputStream).

• Pipes are synchronized communication channels between threads. A pipe is established between two threads. One thread sends data to another by writing to a PipedOutputStream (a subclass of OutputStream). The target thread reads information from the pipe via a PipedInputStream (a subclass of InputStream).

• A PrintStream (a subclass of FilterOutputStream) performs output to the screen (or the “standard output” as defined by your local operating system). System.out is a PrintStream (as is System.err).

• A FilterInputStream filters an InputStream; a FilterOutputStream filters an OutputStream. Filtering means simply that the filter stream provides additional functionality, such as buffering, monitoring of line numbers or aggregating of data bytes into meaningful primitive-data-type units.

• Reading data as raw bytes is fast but crude. Usually, programs read data as aggregates of bytes that form an int, a float, a double, and so on. To accomplish this, we use a DataInputStream (a subclass of class FilterInputStream).

• Interface DataInput is implemented by class DataInputStream and class RandomAccessFile; each needs to read primitive data types from a stream.

• DataInputStreams enable a program to read binary data from an InputStream.

• The DataInput interface includes methods read (for byte arrays), readBoolean, readByte, readChar, readDouble, readFloat, readFully (for byte arrays), readInt, readLine, readLong, readShort, readUnsignedByte, readUnsignedShort, readUTF (for Unicode) and skipBytes.

• The DataOutput interface is implemented by class DataOutputStream (a subclass of class FilterOutputStream) and class RandomAccessFile; each needs to write primitive data types to an OutputStream.

• DataOutputStreams enable a program to write binary data to an OutputStream. The DataOutput interface includes methods flush, size, write (for a byte), write (for a byte array), writeBoolean, writeByte, writeBytes, writeChar, writeChars (for Unicode Strings), writeDouble, writeFloat, writeInt, writeLong, writeShort and writeUTF.

• Buffering is an I/O-performance-enhancement technique.

• With a BufferedOutputStream (a subclass of class FilterOutputStream), each output statement does not necessarily result in an actual physical transfer of data to the output device. Rather, each output operation is directed to a region in memory called a buffer that is large enough to hold the data of many output operations. Then, actual output to the output device is performed
in one large physical output operation each time the buffer fills. The output operations directed to
the output buffer in memory are often called logical output operations.

• With a `BufferedInputStream`, many “logical” chunks of data from a file are read as one
large physical input operation into a memory buffer. As a program requests each new chunk of
data, it is taken from the buffer (this is sometimes referred to as a logical input operation). When
the buffer is empty, the next physical input operation from the input device is performed to read
in the next group of “logical” chunks of data. Thus, the number of physical input operations is
small compared with the number of read requests issued by the program.

• With a `BufferedOutputStream` a partially filled buffer can be forced out to the device at any
time with an explicit `flush`.

• The `ObjectInput` interface is similar to the `DataInput` interface, but includes additional
methods to read `Objects` from `InputStreams`.

• The `ObjectOutput` interface is similar to the `DataOutput` interface, but includes additional
methods to write `Objects` to `OutputStreams`.

• The `ObjectInputStream` and `ObjectOutputStream` classes implement the `Object-
Input` and `ObjectOutput` interfaces, respectively.

• A `PushbackInputStream` is a subclass of class `FilterInputStream`. The application
reading a `PushbackInputStream` reads bytes from the stream and forms aggregates consisting
of several bytes. Sometimes, to determine that one aggregate is complete, the application must
read the first character “past the end” of the first aggregate. Once the program has determined that
the current aggregate is complete, the extra character is “pushed back” onto the stream.

• `PushbackInputStreams` are used by programs, like compilers, that parse their inputs—that
is, break them into meaningful units (such as the keywords, identifiers and operators that the Java
compiler must recognize).

• A `RandomAccessFile` (a subclass of `Object`) is useful for such direct-access applications as
transaction-processing applications, like airline-reservations systems and point-of-sale systems.

• With a sequential-access file, each successive input/output request reads or writes the next consec-
utive set of data in the file.

• With a random-access file, each successive input/output request might be directed to any part of
the file, perhaps one widely separated from the part of the file referenced in the previous request.

• Direct-access applications provide rapid access to specific data items in large files; such applica-
tions are often used while people are waiting for answers—these answers must be made available
quickly, or the people might become impatient and “take their business elsewhere.”

• A `ByteArrayInputStream` (a subclass of `abstract` class `InputStream`) performs its
inputs from a `byte` array in memory.

• A `ByteArrayOutputStream` (a subclass of `abstract` class `OutputStream`) outputs to
a `byte` array in memory.

• An application of `byte`-array input/output is data validation. A program can input an entire line
at a time from the input stream into a `byte` array. Then, a validation routine can scrutinize the
contents of the `byte` array and correct the data, if necessary. The program can now proceed to
input from the `byte` array, knowing that the input data is in the proper format.

• A `StringBufferInputStream` (a subclass of `abstract` class `InputStream`) inputs
from a `StringBuffer` object.

• A `SequenceInputStream` (a subclass of `abstract` class `InputStream`) enables several
`InputStreams` to be concatenated so that the program will see the group as one continuous `Input-
Stream`. As the end of each input stream is reached, the stream is closed, and the next stream
in the sequence is opened.
• Class **BufferedReader** and class **BufferedWriter** enable efficient buffering for character-based streams.

• Class **CharArrayReader** and class **CharArrayWriter** read and write a stream of characters to a character array.

• A **PushbackReader** (a subclass of abstract class **FilterReader**) enables characters to be placed back on a character stream. A **LineNumberReader** (a subclass of **BufferedReader**) is a buffered character-stream that keeps track of line numbers (e.g., a newline, a return or a carriage-return line-feed combination).

• Class **FileReader** (a subclass of **InputStreamReader**) and class **FileWriter** (a subclass of **OutputStreamWriter**) read and write characters to a file. Class **PipedReader** and class **PipedWriter** are piped-character streams. Class **StringReader** and **StringWriter** read and write characters to **Strings**. A **PrintWriter** writes characters to a stream.

• Class **File** enables programs to obtain information about a file or directory.

• Files are opened for output by creating a **FileOutputStream** class object. One argument is passed to the constructor—the filename. Existing files are truncated, and all data in the file is lost. Nonexistent files are created.

• A program can process no files, one file or several files. Each file has a unique name and is associated with an appropriate file stream object. All file-processing methods must refer to a file with the appropriate object.

• A file-position pointer indicates the position in the file from which the next input is to occur or at which the next output is to be placed.

• A convenient way to implement random-access files is by using only fixed-length records. Using this technique, a program can quickly calculate the exact location of a record relative to the beginning of the file.

• Data can be inserted in a random-access file without destroying other data in the file. Data can be updated or deleted without rewriting the entire file.

• The **RandomAccessFile** class has the same capabilities for input and output as the **DataInputStream** and **DataOutputStream** classes and also supports seeking to a specific byte position in the file, with method **seek**.

**TERMINOLOGY**

| absolute path | canWrite method of **File** class |
| alphabetic field | chaining stream objects |
| alphanumeric field | character field |
| binary digit | character set |
| bit | **CharArrayReader** class |
| buffer | **CharArrayWriter** class |
| **BufferedInputStream** class | close a file |
| **BufferedOutputStream** class | **close** method |
| **BufferedReader** class | data hierarchy |
| **BufferedWriter** class | data validation |
| buffering | database |
| byte | database management system (DBMS) |
| **ByteArrayInputStream** class | **DataInput** interface |
| **ByteArrayOutputStream** class | **DataOutput** interface |
| **CANCEL_OPTION** constant | decimal digit |
| canRead method of **File** class | direct-access applications |
Directories only constant
directory
end-of-file
end-of-file marker
EndOfFileException
exists method of File class
field
directory
OutputStream class
field
File class
file
FileOutputStream class
file name
FileReader class
FileInputStream class
PdfReader class
flush
getAbsolutePath method of File class
getName method of File class
getParent method of File class
getSelectedFile method
input stream
InputStream class
InputStreamReader class
error stream
output stream
PdfReader class
PdfOutputStream class
PdfWriter class
PrintStream class
PrintWriter class
PushbackInputStream class
PushbackReader class
r file open mode
random-access file
RandomAccessFile class
read method
readBoolean method
readByte method
readChar method
readDouble method
Reader class
readFloat method
readFully method
readInt method
readLong method
readObject method
readShort method
readUnsignedByte method
readUnsignedShort method
record
record key
relative path
root directory
rw file open mode
seek method
SequenceInputStream class
Serializable interface
setFileSelectionMode method
showOpenDialog method
showSaveDialog method
standard output
StreamReader class
StreamWriter class
System.err (standard error stream)
System.in (standard input stream)
System.out (standard output stream)
SELF-REVIEWS EXERCISES

16.1 Fill in the blanks in each of the following statements:
 a) Ultimately, all data items processed by a computer are reduced to combinations of _______ and _______.
 b) The smallest data item a computer can process is called a _______.
 c) A _______ is a group of related records.
 d) Digits, letters and special symbols are referred to as _______.
 e) A group of related files is called a _______.
 g) RandomAccessFile method _______ reads an integer from the specified stream.
 h) RandomAccessFile method _______ reads a line of text from the specified stream.
 i) RandomAccessFile method _______ sets the file-position pointer to a specific location in a file for input or output.

16.2 State which of the following are true and which are false. If false, explain why.
 a) The programmer must explicitly create the System.in, System.out and System.err objects.
 b) If the file-position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
 c) It is not necessary to search through all the records in a random-access file to find a specific record.
 d) Records in random-access files must be of uniform length.
 e) Method seek must seek relative to the beginning of a file.

16.3 Assume that each of the following statements applies to the same program.
 a) Write a statement that opens file "oldmast.dat" for input; use ObjectInputStream object inOldMaster chained to a FileInputStream object.
 b) Write a statement that opens file "trans.dat" for input; use ObjectInputStream object inTransaction chained to a FileInputStream object.
 c) Write a statement that opens file "newmast.dat" for output (and creation); use ObjectOutputStream object outNewMaster chained to a FileOutputStream.
 d) Write a set of statements that read a record from the file "oldmast.dat". The record consists of integer accountNumber, string name and floating-point currentBalance; use ObjectInputStream object inOldMaster.
 e) Write a set of statements that read a record from the file "trans.dat". The record consists of integer accountNumber and floating-point dollarAmount; use ObjectInputStream object inTransaction.
 f) Write a set of statements that outputs a record to the file "newmast.dat". The record consists of integer accountNumber, string name and floating point currentBalance; use DataOutputStream object outNewMaster.
16.4 Find the error and show how to correct it in each of the following.

a) Assume `account`, `company` and `amount` are declared.

```java
ObjectOutputStream outputStream;
outputStream.writeInt(account);
outputStream.writeChars(company);
outputStream.writeDouble(amount);
```

b) The following statement should read a record from the file "payables.dat". The `ObjectInputStream` object `inPayable` refers to this file, and `FileInputStream` object `inReceivable` refers to the file "receivables.dat".

```java
account = inReceivable.readInt();
companyID = inReceivable.readLong();
amount = inReceivable.readDouble();
```

**ANSWERS TO SELF-REVIEW EXERCISES**

16.1 a) 1s, 0s.  b) bit.  c) file.  d) characters.  e) database.  f) `close`.  g) `readInt`.  h) `readLine`.  i) `seek`.
16.2 a) False. These three streams are created automatically for the programmer.  
b) True.  
c) True.  
d) False. Records in a random-access file are normally of uniform length.  
e) True.
16.3 a) `ObjectInputStream` inOldMaster;

```java
inOldMaster = new ObjectInputStream(  
    new FileInputStream( "oldmast.dat" ) );
```

b) `ObjectInputStream` inTransaction;

```java
inTransaction = new ObjectInputStream(  
    new FileInputStream( "trans.dat" ) );
```

c) `ObjectOutputStream` outNewMaster;

```java
outNewMaster = new ObjectOutputStream(  
    new FileOutputStream( "newmast.dat" ) );
```

d) `accountNumber = inOldMaster.readInt();`

```java
name = inOldMaster.readUTF();
currentBalance = inOldMaster.readDouble();
```

e) `accountNumber = inTransaction.readInt();`

```java
dollarAmount = inTransaction.readDouble();
```

f) `outNewMaster.writeInt( accountNumber );`

```java
outNewMaster.writeUTF( name );
outNewMaster.writeDouble( currentBalance );
```

16.4 a) Error: The file has not been opened before the attempt is made to output data to the stream.  
Correction: Create a new `ObjectOutputStream` object chained to a `FileOutputStream` object to open the file for output.

b) Error: The incorrect `FileInputStream` object is being used to read a record from file "payables.dat".  
Correction: Use object `inPayable` to refer to "payables.dat".
EXERCISES

16.5 Fill in the blanks in each of the following:
   a) Computers store large amounts of data on secondary storage devices as __________.
   b) A __________ is composed of several fields.
   c) A field that may contain only digits, letters and blanks is called an ________ field.
   d) To facilitate the retrieval of specific records from a file, one field in each record is chosen as a ________.
   e) The vast majority of information stored in computer systems is stored in ________ files.
   f) The standard stream objects are __________, __________ and __________.

16.6 State which of the following are true and which are false. If false, explain why.
   a) The impressive functions performed by computers essentially involve the manipulation of zeros and ones.
   b) People specify programs and data items as characters; computers then manipulate and process these characters as groups of zeros and ones.
   c) A person’s 5-digit zip code is an example of a numeric field.
   d) A person’s street address is generally considered to be an alphabetic field.
   e) Data items represented in computers form a data hierarchy in which data items become larger and more complex as we progress from fields to characters to bits, etc.
   f) A record key identifies a record as belonging to a particular field.
   g) Companies store all their information in a single file to facilitate computer processing.
   h) When a program creates a file, the file is automatically retained by the computer for future reference.

16.7 Exercise 16.3 asked the reader to write a series of single statements. Actually, these statements form the core of an important type of file-processing program, namely, a file-matching program. In commercial data processing, it is common to have several files in each application system. In an accounts receivable system, for example, there is generally a master file containing detailed information about each customer, such as the customer’s name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

   a) As transactions occur (i.e., sales are made and cash payments arrive in the mail), they are entered into a file. At the end of each business period (i.e., a month for some companies, a week for others, and a day in some cases) the file of transactions (called "trans.dat" in Exercise 16.3) is applied to the master file (called "oldmast.dat" in Exercise 16.3), thus updating each account’s record of purchases and payments. During an updating run, the master file is rewritten as a new file ("newmast.dat"), which is then used at the end of the next business period to begin the updating process again.

   b) File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period; therefore, no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments could have just moved to this community, and the company might not have had a chance to create a master record for this customer.

   c) Use the statements in Exercise 16.3 as a basis for writing a complete file-matching accounts receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential file with records stored in increasing account-number order.

   d) When a match occurs (i.e., records with the same account number appear on both the master file and the transaction file), add the dollar amount on the transaction file to the
current balance on the master file, and write the "newmast.dat" record. (Assume that purchases are indicated by positive amounts on the transaction file, payments by negative amounts.) When there is a master record for a particular account but no corresponding transaction record, merely write the master record to "newmast.dat". When there is a transaction record but no corresponding master record, print the message "Unmatched transaction record for account number ..." (fill in the account number from the transaction record).

16.8 After writing the program of Exercise 16.7, write a simple program to create some test data for checking out the program. Use the sample account data in Fig. 16.23 and Fig. 16.24. Run the program of Exercise 16.7, using the files of test data created in this exercise. Print the new master file. Check that the accounts have been updated correctly.

16.9 It is possible (actually common) to have several transaction records with the same record key. This occurs because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts receivable file-matching program of Exercise 16.7 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of Exercise 16.8 to include the additional transaction records in Fig. 16.25.

<table>
<thead>
<tr>
<th>Master file</th>
<th>Account number</th>
<th>Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>Alan Jones</td>
<td>348.17</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>Mary Smith</td>
<td>27.19</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>Sam Sharp</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>Suzy Green</td>
<td>-14.22</td>
</tr>
</tbody>
</table>

Fig. 16.23 Sample data for master file.

<table>
<thead>
<tr>
<th>Transaction file</th>
<th>Account number</th>
<th>Transaction amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>27.14</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>62.11</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>100.56</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>82.17</td>
</tr>
</tbody>
</table>

Fig. 16.24 Sample data for transaction file.

<table>
<thead>
<tr>
<th>Account number</th>
<th>Dollar amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>83.89</td>
</tr>
<tr>
<td>700</td>
<td>80.78</td>
</tr>
<tr>
<td>700</td>
<td>1.53</td>
</tr>
</tbody>
</table>

Fig. 16.25 Additional transaction records.
16.10  You are the owner of a hardware store and need to keep an inventory that can tell you what different tools you have, how many of each you have on hand and the cost of each one. Write a program that initializes the random-access file "hardware.dat" to one hundred empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update any information in the file. The tool identification number should be the record number. Use the information in Fig. 16.26 to start your file.

16.11  (Telephone Number Word Generator) Standard telephone keypads contain the digits 0 through 9. The numbers 2 through 9 each have three letters associated with them (see Fig. 16.27).

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in Fig. 16.27 to develop the seven-letter word “NUMBERS.” Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its takeout business could surely do so with the number 825-3688 (i.e., “TAKEOUT”).

<table>
<thead>
<tr>
<th>Record #</th>
<th>Tool name</th>
<th>Quantity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Electric sander</td>
<td>18</td>
<td>35.99</td>
</tr>
<tr>
<td>19</td>
<td>Hammer</td>
<td>128</td>
<td>10.00</td>
</tr>
<tr>
<td>26</td>
<td>Jig saw</td>
<td>16</td>
<td>14.25</td>
</tr>
<tr>
<td>39</td>
<td>Lawn mower</td>
<td>10</td>
<td>79.50</td>
</tr>
<tr>
<td>56</td>
<td>Power saw</td>
<td>8</td>
<td>89.99</td>
</tr>
<tr>
<td>76</td>
<td>Screwdriver</td>
<td>236</td>
<td>4.99</td>
</tr>
<tr>
<td>81</td>
<td>Sledge hammer</td>
<td>32</td>
<td>19.75</td>
</tr>
<tr>
<td>88</td>
<td>Wrench</td>
<td>65</td>
<td>6.48</td>
</tr>
</tbody>
</table>

Fig. 16.26  Data for Exercise 16.10.

<table>
<thead>
<tr>
<th>Digit</th>
<th>Letters</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>A B C</td>
</tr>
<tr>
<td>3</td>
<td>D E F</td>
</tr>
<tr>
<td>4</td>
<td>G H I</td>
</tr>
<tr>
<td>5</td>
<td>J K L</td>
</tr>
<tr>
<td>6</td>
<td>M N O</td>
</tr>
<tr>
<td>7</td>
<td>P R S</td>
</tr>
<tr>
<td>8</td>
<td>T U V</td>
</tr>
<tr>
<td>9</td>
<td>W X Y</td>
</tr>
</tbody>
</table>

Fig. 16.27  Telephone keypad digits and letters.
Each seven-letter phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of letters. It is possible, however, that the owner of a barber shop would be pleased to know that the shop’s telephone number, 424-7288, corresponds to “HAIRCUT.” The owner of a liquor store would, no doubt, be delighted to find that the store’s number, 233-7226, corresponds to “BEERCAN.” A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters “PETCARE.” An automotive dealership would be pleased to know that the dealership number, 639-2277, corresponds to “NEWCARS.”

Write a program that, given a seven-digit number, writes to a file every possible seven-letter word combination corresponding to that number. There are 2187 \( (3^7) \) such combinations. Avoid phone numbers with the digits 0 and 1.
Objectives

• To understand Java networking with URIs, sockets and datagrams.
• To implement Java networking applications by using sockets and datagrams.
• To understand how to implement Java clients and servers that communicate with one another.
• To understand how to implement network-based collaborative applications.
• To construct a multithreaded server.

If the presence of electricity can be made visible in any part of a circuit, I see no reason why intelligence may not be transmitted instantaneously by electricity.
Samuel F. B. Morse

Mr. Watson, come here, I want you.
Alexander Graham Bell

What networks of railroads, highways and canals were in another age, the networks of telecommunications, information and computerization … are today.
Bruno Kreisky, Austrian Chancellor

Science may never come up with a better office-communication system than the coffee break.
Earl Wilson

It’s currently a problem of access to gigabits through punybaud.
J. C. R. Licklider
Chapter 17 Networking

17.1 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 enable applets and applications to communicate with one another (subject to security constraints).

Networking is a massive and complex topic. Computer science and computer engineering students will 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 Java How to Program, Fourth Edition, we introduce a portion of Java’s networking concepts and capabilities. For more advanced networking capabilities, refer to our book Advanced Java 2 Platform How to Program.

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. The classes and interfaces of package *java.net* also offer *packet-based communications* that enable individual *packets* of information to be transmitted—this is commonly used to transmit audio and video over the Internet. In this chapter, we show how to create and manipulate sockets and how to communicate with packets of data.

Our discussion of networking 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. 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 a 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 an appropriate HTML Web page.

We demonstrate the Swing GUI component *JEditorPane* and its ability to render an HTML document downloaded from the World Wide Web. We also introduce Java’s a *socket-based communications*, which enable applications to view networking as if it were file I/O—a program can read from a *socket* or write to a socket as simply as reading from a file or writing to a file. We show how to create and manipulate sockets.

Java provides *stream sockets* and *datagram sockets*. With *stream sockets*, a process establishes a *connection* to another process. While the connection is in place, data flows between the processes in continuous *streams*. Stream sockets are said to provide a *connection-oriented service*. The protocol used for transmission is the popular *TCP* (*Transmission Control Protocol)*.

With *datagram sockets*, individual *packets* of information are transmitted. This is not the right protocol for everyday users, because, unlike TCP, the protocol used—*UDP*, the *User Datagram Protocol*—is a *connectionless service*, and it does not guarantee that packets arrive in any particular order. In fact, packets can be lost, can be duplicated and can even arrive out of sequence. So, with UDP, significant extra programming is required on the user’s part to deal with these problems (if the user chooses to do so). UDP is most appropriate for network applications that do not require the error checking and reliability of TCP. Stream sockets and the TCP protocol will be the most desirable for the vast majority of Java programmers.

**Performance Tip 17.1**

*Connectionless services generally offer greater performance, but less reliability than connection-oriented services.*

**Portability Tip 17.1**

*The TCP protocol and its related set of protocols enable a great variety of heterogeneous computer systems (i.e., computer systems with different processors and different operating systems) to intercommunicate.*

The chapter ends with a case study in which we implement a client/server chat application similar to the instant-messaging services popular on the Web today. The program incorporates many networking techniques introduced in this chapter. The program also introduces *multicasting*, in which a server can *publish* information and clients can *subscribe* to that information. Each time the server publishes more information, all subscribers receive that information. Throughout the examples of this chapter, we will see that many of the networking details are handled by the Java classes we use.
17.2 Manipulating URIs

The Internet offers many protocols. The HyperText Transfer Protocol (HTTP) that forms the basis of the World Wide Web uses URIs (Uniform Resource Identifiers) to locate data on the Internet. URIs frequently are called URLs (Uniform Resource Locators). Actually, a URL is a type of URI. Common URIs refer to files or directories and can reference objects that perform complex tasks, such as database lookups and Internet searches. If you know the URI of publicly available HTML files anywhere on the World Wide Web, you can access that data through HTTP. Java makes it easy to manipulate URIs. Using a URI that refers to the exact location of a resource (such as a Web page) as an argument to the `showDocument` method of interface `AppletContext` causes the browser in which the applet is executing to display the resource at the specified URI. The applet of Fig. 17.1 and Fig. 17.2 enables the user to select a Web page from a `JList` and causes the browser to display the corresponding page.

This applet takes advantage of `applet parameters` specified in the HTML document that invokes the applet. When browsing the World Wide Web, often you will come across applets that are in the public domain—you can use them free of charge on your own Web pages (normally in exchange for crediting the applet’s creator). One common feature of such applets is the ability to customize the applet via parameters that are supplied from the HTML file that invokes the applet. For example, Fig. 17.1 contains the HTML that invokes the applet `SiteSelector` in Fig. 17.2. The HTML document contains eight parameters specified with the `param` element—these lines must appear between the starting and ending `applet` tags. The applet can read these values and use them to customize itself. Any number of `param` tags can appear between the starting and ending `applet` tags. Each parameter has a `name` and a `value`. Applet method `getParameter` retrieves the `value` associated with a specific parameter and returns the `value` as a `String`. The argument passed to `getParameter` is a `String` containing the name of the parameter in the `param` tag. In this example, parameters represent the title of each Web site the user can select and the location of each site. Any number of parameters can be specified. However, these parameters must be named `title#`, where the value of # starts at 0 and increments by one for each new title. Each title should have a corresponding location parameter of the form `location#`, where the value of # starts at 0 and increments by one for each new location. The statement

```
<html>
<title>Site Selector</title>
<body>
  <applet code = "SiteSelector.class" width = "300" height = "75">
    <param name = "title0" value = "Java Home Page">
    <param name = "location0" value = "http://java.sun.com/">
    <param name = "title1" value = "Deitel">
    <param name = "location1" value = "http://www.deitel.com/">
    <param name = "title2" value = "JGuru">
    <param name = "location2" value = "http://www.jGuru.com/">
    <param name = "title3" value = "JavaWorld">
    <param name = "location3" value = "http://www.javaworld.com/">
  </applet>
</body>
</html>
```

Fig. 17.1 HTML document to load `SiteSelector` applet.
String title = getParameter("title0");

gets the value associated with the "title0" parameter and assigns it to String reference title. If there is not a param tag containing the specified parameter, getParameter returns null.

The applet (Fig. 17.2) obtains from the HTML document (Fig. 17.1) the choices that will be displayed in the applet’s JList. Class SiteSelector uses a Hashtable (package java.util) to store the World Wide Web site names and URIs. A Hashtable stores key/value pairs. The program uses the key to store and retrieve the associated value in the Hashtable. In this example, the key is the String in the JList that represents the Web site name, and the value is a URL object that stores the URI of the Web site to display in the browser. Class Hashtable provides two methods of importance in this example—put and get. Method put takes two arguments—a key and its associated value—and places the value in the Hashtable at a location determined by the key. Method get takes one argument—a key—and retrieves the value (as an Object reference) associated with the key. Class SiteSelector also contains a Vector (package java.util) in which the site names are placed so they can be used to initialize the JList (one version of the JList constructor receives a Vector object). A Vector is a dynamically resizable array of Objects. Class Vector provides method add to add a new element to the end of the Vector. Classes Hashtable and Vector are discussed in detail in Chapter 20.

Lines 23–24 in method init (lines 20–63) create the Hashtable and Vector objects. Line 27 calls our utility method getSitesFromHTMLParameters (lines 66–108) to obtain the HTML parameters from the HTML document that invoked the applet.

In method getSitesFromHTMLParameters, line 75 uses Applet method getParameter to obtain a Web site title. If the title is not null, the loop at lines 78–106 begins executing. Line 81 uses Applet method getParameter to obtain the corresponding location. Line 87 uses the location as the initial value of a new URL object. The URL constructor determines whether the String passed as an argument represents a valid Uniform Resource Identifier. If not, the URL constructor throws a MalformedURLException. Notice that the URL constructor must be called in a try block. If the URL constructor generates a MalformedURLException, the call to printStackTrace (line 98) causes the program to display a stack trace. Then the program attempts to obtain the next Web site title. The program does not add the site for the invalid URI to the Hashtable, so the title will not be displayed in the JList.

Common Programming Error 17.1

A MalformedURLException is thrown when a String that is not in proper URI format is passed to a URL constructor.

For a proper URL, line 90 places the title and URL into the Hashtable, and line 93 adds the title to the Vector. Line 104 gets the next title from the HTML document. When the call to getParameter at line 104 returns null, the loop terminates.

When method get SitesFromHTMLParameters returns to init, lines 30–61 construct the applet’s GUI. Lines 31–32 add the JLabel “Choose a site to browse” to the NORTH of the content pane’s BorderLayout. Lines 36–58 register an instance of an anonymous inner class that implements ListSelectionListener to handle the siteChooser’s events. Lines 60–61 add siteChooser to the CENTER of the content pane’s BorderLayout.
// Fig. 17.2: SiteSelector.java
// This program uses a button to load a document from a URL.

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

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

public class SiteSelector extends JApplet {

private Hashtable sites;    // site names and URLs
private Vector siteNames;   // site names
private JList siteChooser;  // list of sites to choose from

public void init()
{

    // create Hashtable and Vector
    sites = new Hashtable();
siteNames = new Vector();

    // obtain parameters from HTML document
    getSiteNamesFromHTMLParameters();

    // create GUI components and layout interface
    Container container = getContentPane();
    container.add( new JLabel( "Choose a site to browse" ),
                   BorderLayout.NORTH );

    siteChooser = new JList( siteNames );
siteChooser.addListSelectionListener(
    new ListSelectionListener() {
    // go to site user selected
    public void valueChanged( ListSelectionEvent event )
    {
        // get selected site name
        Object object = siteChooser.getSelectedValue();

        // use site name to locate corresponding URL
        URL newDocument = ( URL ) sites.get( object );

        // get reference to applet container
        AppletContext browser = getAppletContext();

        // tell applet container to change pages
        browser.showDocument( newDocument );
    }});
}

// read HTML parameters and set up GUI

54     }   // end method valueChanged
55
56 }   // end anonymous inner class
57
58   ); // end call to addListSelectionListener
59
60   container.add( new JScrollPane( siteChooser ),
61       BorderLayout.CENTER );
62
63 }   // end method init
64
65 // obtain parameters from HTML document
66 private void getSitesFromHTMLParameters()
67 {
   // look for applet parameters in the HTML document
   // and add sites to Hashtable
   String title, location;
   URL url;
   int counter = 0;

   // obtain first site title
   title = getParameter( "title" + counter );

   // loop until no more parameters in HTML document
   while ( title != null ) {

      // obtain site location
      location = getParameter( "location" + counter );

      // place title/URL in Hashtable and title in Vector
      try {

         // convert location to URL
         url = new URL( location );

         // put title/URL in Hashtable
         sites.put( title, url );

         // put title in Vector
         siteNames.add( title );

      } catch ( MalformedURLException urlException ) {
         urlException.printStackTrace();
      }

      // process invalid URL format
      ++counter;

      // obtain next site title
      title = getParameter( "title" + counter );

   }   // end while

Fig. 17.2 Loading a document from a URL into a browser (part 2 of 3).
When the user selects one of the Web sites in `siteChooser`, the program calls method `valueChanged` (lines 41–54). Line 44 obtains the selected site name from the `JList`. Line 47 passes the selected site name (the `key`) to `Hashtable` method `get`, which locates and returns an `Object` reference to the corresponding `URL` object (the `value`). The `URL` cast operator converts the reference to a `URL` that can be assigned to reference `newDocument`.

Line 50 uses `Applet` method `getAppletContext` to get a reference to an `AppletContext` object that represents the applet container. Line 53 uses the `AppletContext` reference `browser` to invoke `AppletContext` method `showDocument`, which receives a `URL` object as an argument and passes it to the `AppletContext` (i.e., the browser). The browser displays in the current browser window the World Wide Web resource associated with that `URL`. In this example, all the resources are HTML documents.

For programmers familiar with `HTML frames`, there is a second version of `AppletContext` method `showDocument` that enables an applet to specify the so-called `target frame` in which to display the World Wide Web resource. The second version of `showDocument` takes two arguments—a `URL` object specifying the resource to display and a `String` representing the target frame. There are some special target frames that can be used as the second argument. The target frame `_blank` results in a new Web browser.
window to display the content from the specified URI. The target frame _self specifies that the content from the specified URI should be displayed in the same frame as the applet (the applet’s HTML page is replaced in this case). The target frame _top specifies that the browser should remove the current frames in the browser window, then display the content from the specified URI in the current window. For more information on HTML and frames, see the World Wide Web Consortium (W3C) Web site

http://www.w3.org

[Note: This applet must be run from a World Wide Web browser, such as Netscape Navigator or Microsoft Internet Explorer, to see the results of displaying another Web page. The appletviewer is capable only of executing applets—it ignores all other HTML tags. If the Web sites in the program contained Java applets, only those applets would appear in the appletviewer when the user selects a Web site. Each applet would execute in a separate appletviewer window. Of the browsers mentioned here, only Netscape Navigator 6 currently supports the features of Java 2. You will need to use the Java Plug-in (discussed in Chapter 3) to execute this applet in Microsoft Internet Explorer or older versions of Netscape Navigator.]

17.3 Reading a File on a Web Server

The application of Fig. 17.3 uses Swing GUI component JEditorPane (from package javax.swing) to display the contents of a file on a Web server. The user inputs the URI in the JTextField at the top of the window, and the program displays the corresponding document (if it exists) in the JEditorPane. Class JEditorPane is able to render both plain text and HTML-formatted text, so this application acts as a simple Web browser. The application also demonstrates how to process HyperlinkEvents when the user clicks a hyperlink in the HTML document. The screen captures in Fig. 17.3 illustrate that the JEditorPane can display both simple text (the first screen) and HTML text (the second screen). The techniques shown in this example also can be used in applets. However, applets are allowed to read files only on the server from which the applet was downloaded.

```java
// Fig. 17.3: ReadServerFile.java
// This program uses a JEditorPane to display the contents of a file on a Web server.

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

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

public class ReadServerFile extends JFrame {
    private JTextField enterField;
    private JEditorPane contentsArea;
}
```

Fig. 17.3 Reading a file by opening a connection through a URL (part 1 of 4)
public ReadServerFile()
{
    super( "Simple Web Browser" );

    Container container = getContentPane();

    // create enterField and register its listener
    enterField = new JTextField( "Enter file URL here" );

    enterField.addActionListener(
        new ActionListener() {
            public void actionPerformed( ActionEvent event ) {
                getThePage( event.getActionCommand() );
            }
        } // end anonymous inner class
    ); // end call to addActionListener

    container.add( enterField, BorderLayout.NORTH );

    // create contentsArea and register HyperlinkEvent listener
    contentsArea = new JEditorPane();
    contentsArea.setEditable( false );

    contentsArea.addHyperlinkListener(
        new HyperlinkListener() {
            public void hyperlinkUpdate( HyperlinkEvent event ) {
                if ( event.getEventType() == HyperlinkEvent.EventType.ACTIVATED )
                    getThePage( event.getURL().toString() );
            }
        } // end anonymous inner class
    ); // end call to addHyperlinkListener

    container.add( new JScrollPane( contentsArea ), BorderLayout.CENTER );

    setSize( 400, 300 );
    setVisible( true );
}
private void getThePage(String location) {
    // change mouse cursor to WAIT_CURSOR
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

    // load document into contentsArea and display location in
    // enterField
    try {
        contentsArea.setPage(location);
        enterField.setText(location);
    }

    // process problems loading document
    catch (IOException ioException) {
        JOptionPane.showMessageDialog(this, "Error retrieving specified URL", "Bad URL", JOptionPane.ERROR_MESSAGE);
    }

    setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
}

// begin application execution
public static void main(String args[]) {
    ReadServerFile application = new ReadServerFile();
    application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

Fig. 17.3 Reading a file by opening a connection through a URL (part 3 of 4)
The application class `ReadServerFile` contains `JTextField enterField`, in which the user enters the URI of the file to read and `JEditorPane contentsArea` to display the contents of the file. When the user presses the `Enter` key in the `JTextField`, the program calls method `actionPerformed` (lines 34–37). Line 36 uses `ActionEvent` method `getActionCommand` to get the `String` the user input in the `JTextField` and passes that string to utility method `getThePage` (lines 73–95).

Lines 76–77 in method `getThePage` use method `setCursor` (inherited into class `JFrame` from class `Component`) to change the mouse cursor to the `wait cursor` (normally, an hourglass or a watch). If the file being downloaded is large, the wait cursor indicates to the user that the program is performing a task and that the user should wait for the task to complete. The static `Cursor` method `getPredefinedCursor` receives an integer indicating the cursor type (`Cursor.WAIT_CURSOR` in this case). See the API documentation for class `Cursor` for a complete list of cursors.

Line 82 uses `JEditorPane` method `setPage` to download the document specified by `location` and display it in the `JEditorPane contents`. If there is an error downloading the document, method `setPage` throws an `IOException`. Also, if an invalid URL is specified, a `MalformedURLException` (a subclass of `IOException`) occurs. If the document loads successfully, line 83 displays the current location in `enterField`.

Lines 93–94 sets the `Cursor` back to `Cursor.DEFAULT_CURSOR` (the default `Cursor`) to indicate that the document download is complete.

Typically, an HTML document contains `hyperlinks`—text, images or GUI components which, when clicked, provide quick access to another document on the Web. If a `JEditorPane` contains an HTML document and the user clicks a hyperlink, the `JEditorPane` generates a `HyperlinkEvent` (package `javax.swing.event`) and notifies all registered `HyperlinkListeners` (package `javax.swing.event`) of that event. Lines 49–63 register a `HyperlinkListener` to handle `HyperlinkEvents`. When a `HyperlinkEvent` occurs, the program calls method `hyperlinkUpdate` (lines 54–
59). Lines 56–57 use HyperlinkEvent method getEventType to determine the type of the HyperlinkEvent. Class HyperlinkEvent contains public inner class EventType that defines three hyperlink event types: ACTIVATED (the user clicked a hyperlink to change Web pages), ENTERED (the user moved the mouse over a hyperlink) and EXITED (the user moved the mouse away from a hyperlink). If a hyperlink was ACTIVATED, line 58 uses HyperlinkEvent method getURL to obtain the URL represented by the hyperlink. Method toString converts the returned URL to a String format that can be passed to utility method getThePage.

Software Engineering Observation 17.1

A JEditorPane generates HyperlinkEvents only if it is uneditable.

17.4 Establishing a Simple Server Using Stream Sockets

The two examples discussed so far use high-level Java networking capabilities to communicate between applications. In those examples, it was not the Java programmer’s responsibility to establish the connection between a client and a server. The first program relied on the Web browser to communicate with a Web server. The second program relied on a JEditorPane to perform the connection. This section begins our discussion of creating your own applications that can communicate with one another.

Establishing a simple server in Java requires five steps. Step 1 is to create a ServerSocket object. A call to the ServerSocket constructor such as

```java
ServerSocket server = new ServerSocket( port, queueLength );
```

registers an available port number and specifies a maximum number of clients that can wait to connect to the server (i.e., the queueLength). The port number is used by clients to locate the server application on the server computer. This often is called the handshake point. If the queue is full, the server refuses client connections. The preceding statement establishes the port where the server waits for connections from clients (a process known as binding the server to the port). Each client will ask to connect to the server on this port.

Programs manage each client connection with a Socket object. After binding the server to a port with a ServerSocket (Step 2), the server listens indefinitely (or blocks) for an attempt by a client to connect. To listen for a client, the program calls ServerSocket method accept, as in

```java
Socket connection = server.accept();
```

This statement returns a Socket object when a connection with a client is established.

Step 3 is to get the OutputStream and InputStream objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to the client via an OutputStream object. The server receives information from the client via an InputStream object. To obtain the streams, the server invokes method getOutputStream on the Socket to get a reference to the OutputStream associated with the Socket and invokes method getInputStream on the Socket to get a reference to the InputStream associated with the Socket.

The OutputStream and InputStream objects can be used to send or receive individual bytes or sets of bytes with the OutputStream method write and the InputStream method read, respectively. Often it is useful to send or receive values of primitive
data types (such as `int` and `double`) or `Serializable` class data types (such as `String`) rather than sending bytes. In this case, we can use the techniques of Chapter 16 to chain other stream types (such as `ObjectOutputStream` and `ObjectInputStream`) to the `OutputStream` and `InputStream` associated with the `Socket`. For example,

```java
ObjectInputStream input =
    new ObjectInputStream( connection.getInputStream() );

ObjectOutputStream output =
    new ObjectOutputStream( connection.getOutputStream() );
```

The beauty of establishing these relationships is that whatever the server writes to the `ObjectOutputStream` is sent via the `OutputStream` and is available at the client’s `InputStream` and whatever the client writes to its `OutputStream` (with a corresponding `ObjectOutputStream`) is available via the server’s `InputStream`.

Step 4 is the `processing` phase, in which the server and the client communicate via the `InputStream` and `OutputStream` objects. In Step 5, when the transmission is complete, the server closes the connection by invoking the `close` method on the `Socket` and on the corresponding streams.

### Software Engineering Observation 17.2
With sockets, network I/O appears to Java programs to be identical to sequential file I/O. Sockets hide much of the complexity of network programming from the programmer.

### Software Engineering Observation 17.3
With Java’s multithreading, we can create multithreaded servers that can manage many simultaneous connections with many clients; this multithreaded-server architecture is precisely what popular network servers use.

### Software Engineering Observation 17.4
A multithreaded server can take the `Socket` returned by each call to `accept` and create a new thread that manages network I/O across that `Socket`, or a multithreaded server can maintain a pool of threads (a set of already existing threads) ready to manage network I/O across the new `Sockets` as they are created.

### Performance Tip 17.2
In high-performance systems in which memory is abundant, a multithreaded server can be implemented to create a pool of threads that can be assigned quickly to handle network I/O across each new `Socket` as it is created. Thus, when the server receives a connection, the server need not incur the overhead of thread creation.

### 17.5 Establishing a Simple Client Using Stream Sockets
Establishing a simple client in Java requires four steps. In Step 1, we create a `Socket` to connect to the server. The `Socket` constructor establishes the connection to the server. For example, the statement

```java
Socket connection = new Socket( serverAddress, port );
```

uses the `Socket` constructor with two arguments—the server’s Internet address (`serverAddress`) and the `port` number. If the connection attempt is successful, this statement returns
a `Socket`. A connection attempt that fails throws an instance of a subclass of `IOException`, so many programs simply catch `IOException`. An `UnknownHostException` occurs when a server address indicated by a client cannot be resolved. A `ConnectException` is thrown when an error occurs while attempting to connect to a server.

In Step 2, the client uses `Socket` methods `getInputStream` and `getOutputStream` to obtain references to the `Socket`’s `InputStream` and `OutputStream`. As we mentioned in the preceding section, often it is useful to send or receive values of primitive data types (such as `int` and `double`) or class data types (such as `String` and `Employee`) rather than sending bytes. If the server is sending information in the form of actual data types, the client should receive the information in the same format. Thus, if the server sends values with an `ObjectOutputStream`, the client should read those values with an `ObjectInputStream`.

Step 3 is the processing phase in which the client and the server communicate via the `InputStream` and `OutputStream` objects. In Step 4, the client closes the connection when the transmission is complete by invoking the `close` method on the `Socket` and the corresponding streams. When processing information sent by a server, the client must determine when the server is finished sending information so the client can call `close` to close the `Socket` connection. For example, the `InputStream` method `read` returns the value –1 when it detects end-of-stream (also called EOF—end-of-file). If an `ObjectInputStream` is used to read information from the server, an `EOFException` occurs when the client attempts to read a value from a stream on which end-of-stream is detected.

### 17.6 Client/Server Interaction with Stream Socket Connections

The applications of Fig. 17.4 and Fig. 17.5 use `stream sockets` to demonstrate a simple `client/server chat application`. The server waits for a client connection attempt. When a client application connects to the server, the server application sends a `String` object (remember that `Strings` are `Serializable`) indicating that the connection was successful to the client. Then the client displays the message. Both the client and the server applications contain `JTextFields`, which allow the user to type a message and send it to the other application. When the client or the server sends the `String “TERMINATE”`, the connection between the client and the server terminates. Then the server waits for the next client to connect. The definition of class `Server` appears in Fig. 17.4. The definition of class `Client` appears in Fig. 17.5. The screen captures showing the execution between the client and the server are shown as part of Fig. 17.5.

Class `Server`’s constructor (lines 25–58) creates the GUI of the application (a `JTextField` and a `JTextArea`). The `Server` object displays its output in a `JTextArea`. When the `main` method (lines 186–194) executes, it creates an instance of class `Server`, specifies the window’s default close operation and calls method `runServer` (defined at lines 61–97).

Method `runServer` does the work of setting up the server to receive a connection and processing the connection when it occurs. The method creates a `ServerSocket` called `server` (line 68) to wait for connections. The `ServerSocket` is set up to listen for a connection from a client at port `5000`. The second argument to the constructor is the number of connections that can wait in a queue to connect to the server (`100` in this example). If the queue is full when a client attempts to connect, the server refuses the connection.
// Fig. 17.4: Server.java
// Set up a Server that will receive a connection
// from a client, send a string to the client,
// and close the connection.

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

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

public class Server extends JFrame {

private JTextField enterField;
private JTextArea displayArea;
private ObjectOutputStream output;
private ObjectInputStream input;
private ServerSocket server;
private Socket connection;
private int counter = 1;

// set up GUI
public Server()
{
    super( "Server" );
    Container container = getContentPane();
    // create enterField and register listener
    enterField = new JTextField();
    enterField.setEnabled( false );
    enterField.addActionListener(
        new ActionListener() {

            // send message to client
            public void actionPerformed( ActionEvent event )
            {
                sendData( event.getActionCommand() );
            }
        }  // end anonymous inner class
    ); // end call to addActionListener
    container.add( enterField, BorderLayout.NORTH );
    // create displayArea
    displayArea = new JTextArea();
}
container.add( new JScrollPane( displayArea ),
        BorderLayout.CENTER );

        setSize( 300, 150 );
        setVisible( true );
    }

    // set up and run server
    public void runServer()
    {
        // set up server to receive connections;
        // process connections
        try {

            // Step 1: Create a ServerSocket.
            server = new ServerSocket( 5000, 100 );

            while ( true ) {

                // Step 2: Wait for a connection.
                waitForConnection();

                // Step 3: Get input and output streams.
                getStreams();

                // Step 4: Process connection.
                processConnection();

                // Step 5: Close connection.
                ++counter;
            }
        }

        // process EOFException when client closes connection
        catch ( EOFException eofException ) {
            System.out.println( "Client terminated connection" );
        }

        // process problems with I/O
        catch ( IOException ioException ) {
            ioException.printStackTrace();
        }

        // wait for connection to arrive, then display connection info
        private void waitForConnection() throws IOException
        {
            displayArea.setText( "Waiting for connection\n" );

            // allow server to accept a connection
            connection = server.accept();
        }

Fig. 17.4 Server portion of a client/server stream-socket connection (part 2 of 4).
106       displayArea.append( "Connection " + counter +
107               " received from: " +
108               connection.getInetAddress().getHostName() );
109    }
110
111    // get streams to send and receive data
112    private void getStreams() throws IOException
113    {
114        // set up output stream for objects
115        output = new ObjectOutputStream(
116            connection.getOutputStream() );
117        // flush output buffer to send header information
118        output.flush();
119        // set up input stream for objects
120        input = new ObjectInputStream(
121            connection.getInputStream() );
122        displayArea.append( "\nGot I/O streams\n" );
123    }
124
125    // process connection with client
126    private void processConnection() throws IOException
127    {
128        // send connection successful message to client
129        String message = "SERVER>>> Connection successful";
130        output.writeObject( message );
131        output.flush();
132        // enable enterField so server user can send messages
133        enterField.setEnabled( true );
134
135        // process messages sent from client
136        do {
137            try {
138                message = ( String ) input.readObject();
139                displayArea.append( "\n" + message );
140                displayArea.setCaretPosition(
141                    displayArea.getText().length() );
142            } catch ( ClassNotFoundException classNotFoundException ) {
143                displayArea.append( "\nUnknown object type received" );
144            }
145            while ( !message.equals( "CLIENT>>> TERMINATE" ) );
146        }
147    }
148
Fig. 17.4 Server portion of a client/server stream-socket connection (part 3 of 4).
Port numbers can be between 0 and 65,535. Many operating systems reserve port numbers below 1024 for system services (such as e-mail and World Wide Web servers). Generally, these ports should not be specified as connection ports in user programs. In fact, some operating systems require special access privileges to use port numbers below 1024.

Line 73 calls method `waitForConnection` (lines 100–110) to wait for a client connection. After the connection is established, line 76 calls method `getStreams` (lines 113–127) to obtain references to the `InputStream` and `OutputStream` for the connection. Line 79 calls method `processConnection` to send the initial connection message to the client and to process all messages received from the client. Line 82 calls method `closeConnection` to terminate the connection with the client.
In method `waitForConnection` (lines 100–110), line 105 uses `ServerSocket` method `accept` to wait for a connection from a client and assigns the resulting `Socket` to `connection`. This method blocks until a connection is received (i.e., the thread in which `accept` is called stops executing until a client connects). Lines 107–109 output the host name of the computer that made the connection. `Socket` method `getInetAddress` returns an `InetAddress` object (package `java.net`) containing information about the client computer. `InetAddress` method `getHostName` returns the host name of the client computer. For example, if the Internet address of the computer is `127.0.0.1`, the corresponding host name would be `localhost`.

Method `getStreams` (lines 113–127) obtains references to the `InputStream` and `OutputStream` of the `Socket` and uses them to initialize an `ObjectInputStream` and an `ObjectOutputStream`, respectively. Notice the call to `ObjectOutputStream` method `flush` at line 120. This statement causes the `ObjectOutputStream` on the server to send a `stream header` to the corresponding client’s `ObjectInputStream`. The stream header contains information such as the version of object serialization being used to send objects. This information is required by the `ObjectInputStream` so it can prepare to receive those objects correctly.

**Software Engineering Observation 17.6**

> When using an `ObjectOutputStream` and `ObjectInputStream` to send and receive objects over a network connection, always create the `ObjectOutputStream` first and `flush` the stream so the client’s `ObjectInputStream` can prepare to receive the data. This is required only for applications that communicate using `ObjectOutputStream` and `ObjectInputStream`.

Line 134 of method `processConnection` (lines 130–157) uses `ObjectOutputStream` method `writeObject` to send the string “SERVER>>> Connection successful” to the client. Line 135 flushes the output stream to ensure that the object is sent immediately; otherwise, the object could be held in an output buffer until more information is available to send.

**Performance Tip 17.3**

Output buffers typically are used to increase the efficiency of an application by sending larger amounts of data fewer times. The input and output components of a computer are typically much slower than the memory of the computer.

The `do/while` structure at lines 141–156 loops until the server receives the message “CLIENT>>> TERMINATE.” Line 145 uses `ObjectInputStream` method `readObject` to read a `String` from the client. Line 146 displays the message in the `JTextArea`. Lines 147–148 use `JTextComponent` method `setCaretPosition` to position the input cursor in the `JTextArea` after the last character in the `JTextArea`. This scrolls the `JTextArea` as text is appended to it.

When the transmission is complete, method `processConnection` returns and the program calls method `closeConnection` (lines 160–167) to close the streams associated with the `Socket` and close the `Socket`. Next, the server waits for the next connection attempt from a client by continuing with line 73 at the beginning of the `while` loop.

When the user of the server application enters a `String` in the `JTextField` and presses the `Enter` key, the program calls method `actionPerformed` (lines 40–43), reads the `String` from the `JTextField` and calls utility method `sendData` (lines 170–183).
Method `sendData` sends the `String` object to the client, flushes the output buffer and appends the same `String` to the `JTextArea` in the server window.

Notice that the `Server` receives a connection, processes the connection, closes the connection and waits for the next connection. A more likely scenario would be a `Server` that receives a connection, sets up that connection to be processed as a separate thread of execution, and waits for new connections. The separate threads that process existing connections can continue to execute while the `Server` concentrates on new connection requests.

Like class `Server`, class `Client`’s (Fig. 17.5) constructor creates the GUI of the application (a `JTextField` and a `JTextArea`). The `Client` object displays its output in a `JTextArea`. When the `main` method (line 175–188) executes, it creates an instance of class `Client`, specifies the window’s default close operation and calls method `runClient` (defined at lines 63–90). In this example, you can execute the client from any computer on the Internet and specify the Internet address or host name of the server computer as a command-line argument to the program. For example,

```
java Client 192.168.1.15
```

connects to the `Server` on the computer with Internet address `192.168.1.15`.

```java
// Fig. 17.5: Client.java
// Set up a Client that will read information sent
// from a Server and display the information.

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

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

public class Client extends JFrame {
    private JTextField enterField;
    private JTextArea displayArea;
    private ObjectOutputStream output;
    private ObjectInputStream input;
    private String message = "";
    private String chatServer;
    private Socket client;

    // initialize chatServer and set up GUI
    public Client( String host )
    {
        super( "Client" );
        // set server to which this client connects
        chatServer = host;
    }
}
```

Fig. 17.5  Demonstrating the client portion of a stream-socket connection between a client and a server (part 1 of 5).
Container container = getContentPane();

// create enterField and register listener
enterField = new JTextField();
enterField.setEnabled( false );

enterField.addActionListener(  
    new ActionListener() {  
        // send message to server  
        public void actionPerformed( ActionEvent event )  
        {  
            sendData( event.getActionCommand() );  
        }  
    }  // end anonymous inner class  
); // end call to addActionListener

container.add( enterField, BorderLayout.NORTH );

// create displayArea
displayArea = new JTextArea();
container.add( new JScrollPane( displayArea ),  
    BorderLayout.CENTER );

setSize( 300, 150 );
setVisible( true );

// connect to server and process messages from server
public void runClient()  
{  
    // connect to server, get streams, process connection  
    try {  
        // Step 1: Create a Socket to make connection  
        connectToServer();  
        // Step 2: Get the input and output streams  
        getStreams();  
        // Step 3: Process connection  
        processConnection();  
        // Step 4: Close connection  
        closeConnection();  
    }  
}

Fig. 17.5 Demonstrating the client portion of a stream-socket connection between a client and a server (part 2 of 5).
// server closed connection
catch (EOFException eofException) {
    System.out.println("Server terminated connection");
}

// process problems communicating with server
catch (IOException ioException) {
    ioException.printStackTrace();
}

// get streams to send and receive data
private void getStreams() throws IOException {
    // set up output stream for objects
    output = new ObjectOutputStream(
        client.getOutputStream() );

    // flush output buffer to send header information
    output.flush();

    // set up input stream for objects
    input = new ObjectInputStream(
        client.getInputStream() );

    displayArea.append("Got I/O streams\n" );
}

// connect to server
private void connectToServer() throws IOException {
    displayArea.setText("Attempting connection\n");

    // create Socket to make connection to server
    client = new Socket(
        InetAddress.getByName(chatServer ), 5000 );

    // display connection information
    displayArea.append("Connected to: " +
        client.getInetAddress().getHostName() );
}

// process connection with server
private void processConnection() throws IOException {
    // enable enterField so client user can send messages
    enterField.setEnabled(true);

    // process messages sent from server
    do {
    
Fig. 17.5  Demonstrating the client portion of a stream-socket connection between a client and a server (part 3 of 5).
// read message and display it
try {
    message = (String) input.readObject();
    displayArea.append("n" + message);
    displayArea.setCaretPosition(displayArea.getText().length());
}

// catch problems reading from server
catch (ClassNotFoundException classNotFoundException) {
    displayArea.append("Unknown object type received");
}

} while (!message.equals("SERVER>>> TERMINATE"));

} // end method process connection

// close streams and socket
private void closeConnection() throws IOException
{
    displayArea.append("nClosing connection");
    output.close();
    input.close();
    client.close();
}

// send message to server
private void sendData(String message)
{
    // send object to server
    try {
        output.writeObject("CLIENT>>> " + message);
        output.flush();
        displayArea.append("nCLIENT>>>" + message);
    }

    // process problems sending object
    catch (IOException ioException) {
        displayArea.append("nError writing object");
    }

    // execute application
    public static void main(String args[])
    {
        Client application;

        if (args.length == 0)
            application = new Client("127.0.0.1");
        else
            application = new Client(args[0]);

        Fig. 17.5 Demonstrating the client portion of a stream-socket connection between a client and a server (part 4 of 5).
Client method `runClient` (lines 63–90) performs the work necessary to connect to the `Server`, to receive data from the `Server` and to send data to the `Server`. Line 69 calls method `connectToServer` (lines 110–121) to perform the connection. After connecting, line 72 calls method `getStreams` (lines 93–107) to obtain references to the `Socket`'s `InputStream` and `OutputStream` objects. Then line 75 calls method `processConnection` (124–147) to handle messages sent from the server. When the connection terminates, line 78 calls `closeConnection` to close the streams and the `Socket`.

Method `connectToServer` (lines 110–121) creates a `Socket` called `client` (lines 115–116) to establish a connection. The method passes two arguments to the
Socket constructor—the Internet address of the server computer and the port number (5000) where that computer is awaiting client connections. The call to InetAddress static method getByName in the first argument returns an InetAddress object containing the Internet address specified as a command-line argument to the application (or 127.0.0.1 if no command-line arguments are specified). Method getByName can receive a String containing either the actual Internet address or the host name of the server. The first argument also could have been written other ways. For the localhost address 127.0.0.1, the first argument could be

InetAddress.getByName( "localhost" )

or

InetAddress.getLocalHost()

Also, there are versions of the Socket constructor that receive a String for the Internet address or host name. The first argument could have been specified as "127.0.0.1" or "localhost". [Note: We chose to demonstrate the client/server relationship by connecting between programs executing on the same computer (localhost). Normally, this first argument would be the Internet address of another computer. The InetAddress object for another computer can be obtained by specifying the Internet address or host name of the other computer as the String argument to InetAddress.getByName.]

The Socket constructor’s second argument is the server port number. This number must match the port number at which the server is waiting for connections (called the handshake point). Once the connection is made, a message is displayed in the JTextArea (lines 119–120) indicating the name of the server computer to which the client connected.

The Client uses an ObjectOutputStream to send data to the server and an ObjectInputStream to receive data from the server. Method getStreams (lines 93–107) creates the ObjectOutputStream and ObjectInputStream objects that use the OutputStream and InputStream objects associated with client.

Method processConnection (lines 124–147) contains a do/while structure that loops until the client receives the message “SERVER>>> TERMINATE.” Line 134 uses ObjectInputStream method readObject to read a String from the server. Line 135 displays the message in the JTextArea. Lines 136–137 use JTextComponent method setCaretPosition to position the input cursor in the JTextArea after the last character in the JTextArea.

When the transmission is complete, method closeConnection (lines 150–156) closes the streams and the Socket.

When the user of the client application enters a String in the JTextField and presses the Enter key, the program calls method actionPerformed (lines 42–45) to read the String from the JTextField and invoke utility method sendData (159–172). Method sendData sends the String object to server client, flushes the output buffer and appends the same String to the JTextArea in the client window.

17.7 Connectionless Client/Server Interaction with Datagrams

We have been discussing connection-oriented, streams-based transmission. Now we consider connectionless transmission with datagrams.
Connection-oriented transmission is like the telephone system in which you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for the duration of your phone call, even when you are not talking.

Connectionless transmission with datagrams is more like the way mail is carried via the postal service. If a large message will not fit in one envelope, you break it into separate message pieces that you place in separate, sequentially numbered envelopes. Each of the letters is then mailed at the same time. The letters could arrive in order, out of order or not at all (although the last case is rare, it does happen). The person at the receiving end reassembles the message pieces into sequential order before attempting to make sense of the message. If your message is small enough to fit in one envelope, you do not have to worry about the “out-of-sequence” problem, but it is still possible that your message might not arrive. One difference between datagrams and postal mail is that duplicates of datagrams can arrive on the receiving computer.

The programs of Fig. 17.6 and Fig. 17.7 use datagrams to send packets of information between a client application and a server application. In the Client application (Fig. 17.7), the user types a message into a JTextField and presses Enter. The program converts the message into a byte array and places it in a datagram packet that is sent to the server. The Server (Fig. 17.6) receives the packet and displays the information in the packet, then echoes the packet back to the client. When the client receives the packet, the client displays the information in the packet. In this example, the Client and Server classes are implemented similarly.

Class Server (Fig. 17.6) defines two DatagramPackets that the server uses to send and receive information and one DatagramSocket that sends and receives these packets. The constructor for class Server (lines 20–41) creates the graphical user interface where the packets of information will be displayed. Next the constructor creates the DatagramSocket in a try block. Line 32 uses the DatagramSocket constructor that takes an integer port number argument (5000) to bind the server to a port where the server can receive packets from clients. Clients sending packets to this Server specify port 5000 in the packets they send. The DatagramSocket constructor throws a SocketException if it fails to bind the DatagramSocket to a port.

Common Programming Error 17.2

Specifying a port that is already in use or specifying an invalid port number when creating a DatagramSocket results in a BindException.

```java
// Fig. 17.6: Server.java
// Set up a Server that will receive packets from a client and send packets to a client.

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

Fig. 17.6 Demonstrating the server side of connectionless client/server computing with datagrams (part 1 of 4).
// Java extension packages
import javax.swing.*;

public class Server extends JFrame {

    private JTextArea displayArea;
    private DatagramPacket sendPacket, receivePacket;
    private DatagramSocket socket;

    // set up GUI and DatagramSocket
    public Server() {
        super( "Server" );

        displayArea = new JTextArea();
        getContentPane().add( new JScrollPane( displayArea ),
                          BorderLayout.CENTER );
        setSize( 400, 300 );
        setVisible( true );

        // create DatagramSocket for sending and receiving packets
        try {
            socket = new DatagramSocket( 5000 );
        }

        // process problems creating DatagramSocket
        catch( SocketException socketException ) {
            socketException.printStackTrace();
            System.exit( 1 );
        }
    }

    // wait for packets to arrive, then display data and echo
    // packet to client
    public void waitForPackets() {
        // loop forever
        while ( true ) {

            // receive packet, display contents, echo to client
            try {

                // set up packet
                byte data[] = new byte[ 100 ];
                receivePacket =
                    new DatagramPacket( data, data.length );

                // wait for packet
                socket.receive( receivePacket );

                // process packet
                displayPacket();
            }
        }
    }

    // display packet contents
    private void displayPacket() {
        displayArea.setText( new String( receivePacket.getData() ) );
    }
}

Fig. 17.6 Demonstrating the server side of connectionless client/server computing
with datagrams (part 2 of 4).
// echo information from packet back to client
sendPacketToClient();

// process problems manipulating packet
try {
    displayArea.append( ioException.toString() + "\n" );
    ioException.printStackTrace();
} catch( IOException ioException ) {
    displayArea.append( ioException.toString() + "\n" );
    ioException.printStackTrace();
}

} // end while

} // end method waitForPackets

// display packet contents
private void displayPacket() {
    displayArea.append("\nPacket received:"
                    + "\nFrom host: " + receivePacket.getAddress() + "\nHost port: " + receivePacket.getPort() + "\nLength: " + receivePacket.getLength() + "\nContaining:\n"
                    + new String( receivePacket.getData(), 0, receivePacket.getLength() ) );
}

// echo packet to client
private void sendPacketToClient() throws IOException {
    displayArea.append("\n\nEcho data to client..."
                    + "\nPacket sent\n" );
    displayArea.setCaretPosition( displayArea.getText().length() );

    // create packet to send
    sendPacket = new DatagramPacket( receivePacket.getData(), receivePacket.getLength(), receivePacket.getAddress(), receivePacket.getPort() );

    // send packet
    socket.send( sendPacket );
    displayArea.append("Packet sent\n" );
}

// execute application
public static void main( String args[] ) {
    Server application = new Server();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

Fig. 17.6 Demonstrating the server side of connectionless client/server computing with datagrams (part 3 of 4).
Server method `waitForPackets()` (lines 45–76) uses an infinite loop to wait for packets to arrive at the `Server`. Lines 54–56 create a `DatagramPacket` in which a received packet of information can be stored. The `DatagramPacket` constructor for this purpose receives two arguments—a `byte` array containing the data and the length of the `byte` array. Line 59 waits for a packet to arrive at the `Server`. Method `receive` blocks until a packet arrives, then stores the packet in its `DatagramPacket` argument. Method `receive` throws an `IOException` if an error occurs receiving a packet.

When a packet arrives, the program calls method `displayPacket` (lines 79–88) to append the packet’s contents to `displayArea`. `DatagramPacket` method `getAddress` (line 82) returns an `InetAddress` object containing the host name of the computer from which the packet was sent. Method `getPort` (line 83) returns an integer specifying the port number through which the host computer sent the packet. Method `getLength` (line 84) returns an integer representing the number of bytes of data that were sent. Method `getData` (line 86) returns a `byte` array containing the data that was sent. The program uses the `byte` array to initialize a `String` object so the data can be output to the `JTextArea`.

After displaying a packet, the program calls method `sendPacketToClient` (line 65) to create a new packet and send it to the client. Lines 96–98 create `sendPacket` and pass four arguments to the `DatagramPacket` constructor. The first argument specifies the `byte` array to send. The second argument specifies the number of bytes to send. The third argument specifies the client computer’s Internet address, to which the packet will be sent. The fourth argument specifies the port where the client is waiting to receive packets. Line 101 sends the packet over the network. Method `send` throws an `IOException` if an error occurs sending a packet.

Class `Client` (Fig. 17.7) works similarly to class `Server`, except that the `Client` sends packets only when the user types a message in a `JTextField` and presses the `Enter`
key. When this occurs, the program calls method `actionPerformed` (lines 34–67), which converts the `String` the user entered in the `JTextField` into a `byte` array (line 45). Lines 48–50 create a `DatagramPacket` and initialize it with the `byte` array, the length of the `String` that was entered by the user, the Internet address to which the packet is to be sent (`InetAddress.getLocalHost()` in this example) and the port number at which the `Server` is waiting for packets. Line 53 sends the packet. Note that the client in this example must know that the server is receiving packets at port 5000; otherwise, the server will not receive the packets.

---

Fig. 17.7: Demonstrating the client side of connectionless client/server computing with datagrams (part 1 of 4).
// get message from textfield and convert to array of bytes
String message = event.getActionCommand();
byte data[] = message.getBytes();

// create sendPacket
sendPacket = new DatagramPacket(
data, data.length,
    InetAddress.getLocalHost(), 5000);

// send packet
socket.send( sendPacket );

displayArea.append( "Packet sent
" );
displayArea.setCaretPosition(
    displayArea.getText().length() );

}  // end actionPerformed

}  // end anonymous inner class

); // end call to addActionListener

container.add( enterField, BorderLayout.NORTH );

displayArea = new JTextArea();
container.add( new JScrollPane( displayArea ),
    BorderLayout.CENTER );

setSize( 400, 300 );
setVisible( true );

// create DatagramSocket for sending and receiving packets
try {
    socket = new DatagramSocket();
}

// catch problems creating DatagramSocket
catch( SocketException socketException ) {
    socketException.printStackTrace();
    System.exit( 1 );
}

}  // end Client constructor

Fig. 17.7 Demonstrating the client side of connectionless client/server computing with datagrams (part 2 of 4).
// wait for packets to arrive from Server,
// then display packet contents
public void waitForPackets()
{
    // loop forever
    while ( true ) {

        // receive packet and display contents
        try {
            // set up packet
            byte data[] = new byte[ 100 ];
            receivePacket =
                new DatagramPacket( data, data.length );

            // wait for packet
            socket.receive( receivePacket );

            // display packet contents
            displayPacket();
        }

        // process problems receiving or displaying packet
        catch( IOException exception ) {
            displayArea.append( exception.toString() + "\n" );
            exception.printStackTrace();
        }
    }  // end while

}  // end method waitForPackets

// display contents of receivePacket
private void displayPacket()
{
    displayArea.append( "\nPacket received:" +
        "\nFrom host: " + receivePacket.getAddress() +
        "\nHost port: " + receivePacket.getPort() +
        "\nLength: " + receivePacket.getLength() +
        "\nContaining:\n\t" +
        new String( receivePacket.getData(), 0,
            receivePacket.getLength() ) );

    displayArea.setCaretPosition(
        displayArea.getText().length() );
}

// execute application
public static void main( String args[] )
{
    Client application = new Client();

Fig. 17.7 Demonstrating the client side of connectionless client/server computing with datagrams (part 3 of 4).
Notice that the `DatagramSocket` constructor call (line 84) in this application does not specify any arguments. This constructor allows the computer to select the next available port number for the `DatagramSocket`. The client does not need a specific port number, because the server receives the client’s port number as part of each `DatagramPacket` sent by the client. Thus, the server can send packets back to the same computer and port number from which the server receives a packet of information.

Client method `waitForPackets` (lines 97–125) uses an infinite loop to wait for packets from the server. Line 111 blocks until a packet arrives. Note that this does not prevent the user from sending a packet, because the GUI events are handled in the event dispatch thread. It only prevents the `while` loop from continuing until a packet arrives at the Client. When a packet arrives, line 111 stores the packet in `receivePacket`, and line 114 calls method `displayPacket` (128–140) to display the packet’s contents in the `JTextArea`.

17.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

In this section, we present the popular game Tic-Tac-Toe implemented by using client/server techniques with stream sockets. The program consists of a `TicTacToeServer` application (Fig. 17.8) that allows two `TicTacToeClient` applets (Fig. 17.9) to connect to the server and play Tic-Tac-Toe (outputs shown in Fig. 17.10). As the server receives each client connection, it creates an instance of inner class `Player` (lines 158–279 of Fig. 17.8) to process the client in a separate thread. These threads enable the clients to play the game independently. The server assigns Xs to the first client to connect (X makes the first move) and assigns...
Os to the second client to connect. The server maintains the information about the board so it can determine whether a player’s move is a valid or invalid move. Each **TicTacToeClient** applet (Fig. 17.9) maintains its own GUI version of the Tic-Tac-Toe board on which it displays the state of the game. The clients can place a mark only in an empty square on the board. Class **Square** (lines 212–270 of Fig. 17.9) implements each of the nine squares on the board. Class **TicTacToeServer** and class **Player** are implemented in file **TicTacToeServer.java** (Fig. 17.8). Class **TicTacToeClient** and class **Square** are implemented in file **TicTacToeClient.java** (Fig. 17.9).

```java
// Fig. 17.8: TicTacToeServer.java
// This class maintains a game of Tic-Tac-Toe for two client applets.

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

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

public class TicTacToeServer extends JFrame {
    private byte board[];
    private JTextArea outputArea;
    private Player players[];
    private ServerSocket server;
    private int currentPlayer;

    public TicTacToeServer()
    {
        super( "Tic-Tac-Toe Server" );

        board = new byte[ 9 ];
        players = new Player[ 2 ];
        currentPlayer = 0;

        // set up ServerSocket
        try {
            server = new ServerSocket( 5000, 2 );
        }
        catch( IOException ioException ) {
            ioException.printStackTrace();
            System.exit( 1 );
        }

        // set up JTextArea to display messages during execution
        outputArea = new JTextArea();
    }
}
```

Fig. 17.8 Server side of client/server Tic-Tac-Toe program (part 1 of 6).
```java
43     getContentPane().add( outputArea, BorderLayout.CENTER );
44     outputArea.setText( "Server awaiting connections\n" );
45
46     setSize( 300, 300 );
47     setVisible( true );
48 }
49
50 // wait for two connections so game can be played
51 public void execute()
52 {
53     // wait for each client to connect
54     for ( int i = 0; i < players.length; i++ ) {
55         // wait for connection, create Player, start thread
56         try {
57             players[ i ] = new Player( server.accept(), i );
58             players[ i ].start();
59         }
60         // process problems receiving connection from client
61         catch( IOException ioException ) {
62             ioException.printStackTrace();
63             System.exit( 1 );
64         }
65     }
66     // Player X is suspended until Player O connects.
67     // Resume player X now.
68     synchronized ( players[ 0 ] ) {
69             players[ 0 ].setSuspended( false );
70             players[ 0 ].notify();
71         }
72     // end method execute
73 }
74
75 // display a message in outputArea
76 public void display( String message )
77 {
78     outputArea.append( message + "\n" );
79 }
80
81 // Determine if a move is valid.
82 // This method is synchronized because only one move can be
83 // made at a time.
84 public synchronized boolean validMove(
85     int location, int player )
86 {
87         boolean moveDone = false;
88         // while not current player, must wait for turn
89         while ( player != currentPlayer ) {
90             synchronized ( players[ location ] ) {
91                 players[ location ].setSuspended( false );
92                 players[ location ].notify();
93             }
94         }
95     }
96 }
97
Fig. 17.8 Server side of client/server Tic-Tac-Toe program (part 2 of 6).
```
// wait for turn
try {
    wait();
}

// catch wait interruptions
catch( InterruptedException interruptedException ) {
    interruptedException.printStackTrace();
}

// if location not occupied, make move
if ( !isOccupied( location ) ) {
    // set move in board array
    board[ location ] = ( byte ) ( currentPlayer == 0 ? 'X' : 'O' );
    // change current player
currentPlayer = ( currentPlayer + 1 ) % 2;
    // let new current player know that move occurred
    players[ currentPlayer ].otherPlayerMoved( location );
    // tell waiting player to continue
    notify();
    // tell player that made move that the move was valid
    return true;
}
else
    return false;

// determine whether location is occupied
public boolean isOccupied( int location )
{
    if ( board[ location ] == 'X' || board[ location ] == 'O' )
        return true;
    else
        return false;
}

// place code in this method to determine whether game over
public boolean gameOver()
{
    return false;
}
// execute application
public static void main( String args[] )
{
    TicTacToeServer application = new TicTacToeServer();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    application.execute();
}

// private inner class Player manages each Player as a thread
private class Player extends Thread {
    private Socket connection;
    private DataInputStream input;
    private DataOutputStream output;
    private int playerNumber;
    private char mark;
    protected boolean suspended = true;

    // set up Player thread
    public Player( Socket socket, int number )
    {
        playerNumber = number;

        // specify player's mark
        mark = ( playerNumber == 0 ? 'X' : 'O' );
        connection = socket;

        // obtain streams from Socket
        try {
            input = new DataInputStream( connection.getInputStream() );
            output = new DataOutputStream( connection.getOutputStream() );
        }
        catch( IOException ioException ) {
            ioException.printStackTrace();
            System.exit( 1 );
        }

        // process problems getting streams
        else {
            throw new RuntimeException( "Could not connect to server" );
        }

        // send message that other player moved; message contains
        // a String followed by an int
        public void otherPlayerMoved( int location )
        { // send message indicating move
            try {
                output.writeUTF( "Opponent moved" );
            }
            catch( IOException ioException ) {
                ioException.printStackTrace();
                System.exit( 1 );
            }
        }
}

Fig. 17.8  Server side of client/server Tic-Tac-Toe program (part 4 of 6).
```java
output.writeInt( location );

// process problems sending message
try {
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }
}

// control thread's execution
public void run() {
    // send client message indicating its mark (X or O),
    // process messages from client
    try {
        display( "Player " + ( playerNumber == 0 ? 'X' : 'O' ) + " connected" );
        // send player's mark
        output.writeChar( mark );
        // send message indicating connection
        output.writeUTF( "Player " +
            ( playerNumber == 0 ? "X connected\n" : "O connected, please wait\n" ) );
    }
    // if player X, wait for another player to arrive
    if ( mark == 'X' ) {
        output.writeUTF( "Waiting for another player" );
        // wait for player 0
        try {
            synchronized( this ) {
                while ( suspended )
                    wait();
            }
        }
        // process interruptions while waiting
        catch ( InterruptedException exception ) {
            exception.printStackTrace();
        }
    // send message that other player connected and
    // player X can make a move
    output.writeUTF( "Other player connected. Your move." );
}

    // while game not over
    while ( ! gameOver() ) {
```

Fig. 17.8  Server side of client/server Tic-Tac-Toe program (part 5 of 6).
We begin with a discussion of the server side of the Tic-Tac-Toe game. When the `TicTacToeServer` application executes, the `main` method (lines 147–155) creates a `TicTacToeServer` object called `application`. The constructor (lines 22–48) attempts to set up a `ServerSocket`. If successful, the program displays the server
window, and main invokes the TicTacToeServer method execute (lines 51–76). Method execute loops twice, blocking at line 58 each time while waiting for a client connection. When a client connects, line 58 creates a new Player object to manage the connection as a separate thread, and line 59 calls that object’s start method to begin executing the thread.

When the TicTacToeServer creates a Player, the Player constructor (lines 167–189) receives the Socket object representing the connection to the client and gets the associated input and output streams. The Player’s run method (lines 208–271) controls the information that is sent to the client and the information that is received from the client. First, it tells the client that the client’s connection has been made (lines 213–214), then it passes to the client the character that the client will place on the board when a move is made (line 217). Lines 230–233 suspend each Player thread as it starts executing, because neither player is allowed to make a move when it first connects. Player X can move only when player O connects, and player O can make a move only after player X.

At this point, the game can be played, and the run method begins executing its while structure (lines 248–260). Each iteration of this while structure reads an integer (line 253) representing the location where the client wants to place a mark, and line 254 invokes the TicTacToeServer method validMove (lines 87–129) to check the move. Lines 254–259 send a message to the client indicating whether the move was valid. The program maintains board locations as numbers from 0 to 8 (0 through 2 for the first row, 3 through 5 for the second row and 6 through 8 for the third row).

Method validMove (lines 87–129 in class TicTacToeServer) is a synchronized method that allows only one player at a time to move. Synchronizing validMove prevents both players from modifying the state information of the game simultaneously. If the Player attempting to validate a move is not the current player (i.e., the one allowed to make a move), the Player is placed in a wait state until it is that Player’s turn to move. If the position for the move being validated is already occupied on the board, validMove returns false. Otherwise, the server places a mark for the player in its local representation of the board (lines 110–111), notifies the other Player object (line 117) that a move has been made (so the client can be sent a message), invokes method notify (line 120) so the waiting Player (if there is one) can validate a move and returns true (line 123) to indicate that the move is valid.

When a TicTacToeClient (Fig. 17.9) applet begins execution, it creates a JTextArea in which messages from the server and a representation of the board using nine Square objects are displayed. The applet’s start method (lines 80–104) opens a connection to the server and gets the associated input and output streams from the Socket object. Class TicTacToeClient implements interface Runnable so that a separate thread can read messages from the server. This approach enables the user to interact with the board (in the event-dispatch thread) while waiting for messages from the server. After establishing the connection to the server, line 102 creates Thread object outputThread and initializes it with the Runnable applet, then line 103 calls the thread’s start method. The applet’s run method (lines 108–137) controls the separate thread of execution. The method first reads the mark character (X or O) from the server (line 112), then loops continually (lines 123–135) and reads messages from the server (line 127). Each message is passed to the applet’s processMessage method (lines 140–211) for processing.
// Fig. 17.9: TicTacToeClient.java
// Client for the TicTacToe program

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

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

// Client class to let a user play Tic-Tac-Toe with
// another user across a network.
public class TicTacToeClient extends JApplet
  implements Runnable {
  private JTextField idField;
  private JTextArea displayArea;
  private JPanel boardPanel, panel2;
  private Square board[][], currentSquare;
  private Socket connection;
  private DataInputStream input;
  private DataOutputStream output;
  private Thread outputThread;
  private char myMark;
  private boolean myTurn;

  // Set up user-interface and board
  public void init()
  {
    Container container = getContentPane();

    // set up JTextArea to display messages to user
    displayArea = new JTextArea( 4, 30 );
    displayArea.setEditable( false );
    container.add( new JScrollPane( displayArea ),
      BorderLayout.SOUTH );

    // set up panel for squares in board
    boardPanel = new JPanel();
    boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );

    // create board
    board = new Square[3][3];

    // When creating a Square, the location argument to the
    // constructor is a value from 0 to 8 indicating the
    // position of the Square on the board. Values 0, 1,
    // and 2 are the first row, values 3, 4, and 5 are the
    // second row. Values 6, 7, and 8 are the third row.
    for ( int row = 0; row < board.length; row++ ) {
      for ( int col = 0; col < board[row].length; col++ ) {
```java
for (int column = 0; column < board[row].length; column++) {
    // create Square
    board[row][column] =
        new Square(' ', row * 3 + column);
    boardPanel.add(board[row][column]);
}

// textfield to display player's mark
idField = new JTextField();
idField.setEditable(false);
container.add(idField, BorderLayout.NORTH);

// set up panel to contain boardPanel (for layout purposes)
panel2 = new JPanel();
panel2.add(boardPanel, BorderLayout.CENTER);
container.add(panel2, BorderLayout.CENTER);

// Make connection to server and get associated streams.
// Start separate thread to allow this applet to
// continually update its output in text area display.
public void start() {
    try {
        // make connection
        connection = new Socket(
            InetAddress.getByName("127.0.0.1"), 5000);

        // get streams
        input = new DataInputStream(
            connection.getInputStream());
        output = new DataOutputStream(
            connection.getOutputStream());
    }
    catch (IOException ioException) {
        ioException.printStackTrace();
    }
    // create and start output thread
    outputThread = new Thread(this);
    outputThread.start();
}
```

Fig. 17.9 Client side of client/server Tic-Tac-Toe program (part 2 of 6).


// control thread that allows continuous update of the
text area displayArea

public void run()
{
  // get player's mark (X or O)
  try {
    myMark = input.readChar();
    idField.setText( "You are player " + myMark + "\n" );
    myTurn = ( myMark == 'X' ? true : false );
  }

  // process problems communicating with server
  catch ( IOException ioException ) {
    ioException.printStackTrace();
  }

  // receive messages sent to client and output them
  while ( true ) {
    // read message from server and process message
    try {
      String message = input.readUTF();
      processMessage( message );
    }

    // process problems communicating with server
    catch ( IOException ioException ) {
      ioException.printStackTrace();
    }
  }
}

// process messages received by client

public void processMessage( String message )
{
  // valid move occurred
  if ( message.equals( "Valid move." ) ) {
    displayArea.append( "Valid move, please wait.\n" );

    // set mark in square from event-dispatch thread
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run()
        {
          currentSquare.setMark( myMark );
        }
      }
    );
  }
}
// invalid move occurred
else if ( message.equals( "Invalid move, try again" ) ) {
    displayArea.append( message + "\n" );
    myTurn = true;
}

// opponent moved
else if ( message.equals( "Opponent moved" ) ) {

    // get move location and update board
    try {
        final int location = input.readInt();

        // set mark in square from event-dispatch thread
        SwingUtilities.invokeLater(
            new Runnable() {
                public void run()
                {
                    int row = location / 3;
                    int column = location % 3;
                    board[ row ][ column ].setMark(
                        ( myMark == 'X' ? 'O' : 'X' ) );
                    displayArea.append( "Opponent moved. Your turn.\n" );
                }
            } // end call to invokeLater
        myTurn = true;
    }

    // process problems communicating with server
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }

    // simply display message
else
    displayArea.append( message + "\n" );

displayArea.setCaretPosition(
    displayArea.getText().length() );
public void sendClickedSquare( int location )
{
    if ( myTurn ) {
        // send location to server
        try {
            output.writeInt( location );
            myTurn = false;
        }
        // process problems communicating with server
        catch ( IOException ioException ) {
            ioException.printStackTrace();
        }
    }
}

// set current Square
public void setCurrentSquare( Square square )
{
    currentSquare = square;
}

// private class for the squares on the board
private class Square extends JPanel {
    private char mark;
    private int location;

    public Square( char squareMark, int squareLocation )
    {
        mark = squareMark;
        location = squareLocation;
        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased( MouseEvent e )
                {
                    setCurrentSquare( Square.this );
                    sendClickedSquare( getSquareLocation() );
                }
            } // end anonymous inner class
        ); // end call to addMouseListener
    }
} // end Square constructor

Fig. 17.9 Client side of client/server Tic-Tac-Toe program (part 5 of 6).
If the message received is **Valid move.**, lines 143–159 display the message **Valid move, please wait.** and call class **Square**’s **setMark** method to set the client’s mark in the current square (the one in which the user clicked) using **SwingUtilities** method **invokeLater** to ensure that the GUI updates occur in the event dispatch thread. If the message received is **Invalid move, try again.**, lines 162–165 display the message so the user can click a different square. If the message received is **Opponent moved.**, lines 168–195 read an integer from the server indicating where the opponent moved and place a mark in that square of the board (again using **SwingUtilities** method **invokeLater** to ensure that the GUI updates occur in the event dispatch thread). If any other message is received, line 206 simply displays the message.
Fig. 17.10 Sample outputs from the client/server Tic-Tac-Toe program (part 1 of 2).
17.9 Security and the Network

As much as we look forward to writing a great variety of powerful network-based applications, our efforts may be crimped by limitations imposed on Java because of security concerns. Many Web browsers, such as Netscape Communicator and Microsoft Internet Explorer, by default prohibit Java applets from doing file processing on the machines on which they execute. Think about it. A Java applet is designed to be sent to your browser via an HTML document that could be downloaded from any Web server in the world. Often you will know very little about the sources of Java applets that will execute on your system. To allow these applets free rein with your files could be disastrous.

A more subtle situation occurs with limiting the machines to which executing applets can connect. To build truly collaborative applications, we would ideally like to have our applets communicate with machines almost anywhere. The Java security manager in a Web browser often restricts an applet so that it can communicate only with the machine from which it was originally downloaded.

These restrictions might seem too harsh. However, the Java Security API now provides capabilities for signed applets that will enable browsers to determine whether an applet is downloaded from a trusted source. In cases where an applet is trusted, the applet can be given additional access to the computer on which the applet is executing. The features of the Java Security API and additional networking capabilities are discussed in our text Advanced Java 2 Platform How to Program.

17.10 DeitelMessenger Chat Server and Client

Chat rooms have become quite common on the Internet. Chat rooms provide a central location where users can chat with each other through short text messages. Each participant in a chat room can see all messages that other users post, and each user can post messages in the chat room. This section presents our capstone networking case study that integrates many of the Java networking, multithreading and Swing GUI features we have learned thus far to build an online chat system. We also introduce multicast, which enables an application to send DatagramPackets to groups of clients. After reading this section, you will be able to build significant networking applications.
17.10.1 DeitelMessengerServer and Supporting Classes

The DeitelMessengerServer (Fig. 17.11) is the heart of the online chat system. Chat clients can participate in a chat by connecting to the DeitelMessengerServer. Method startServer (lines 19–54) launches DeitelMessengerServer. Lines 25–26 create a ServerSocket to accept incoming network connections. Recall that the ServerSocket constructor takes as its first argument the port on which the server should listen for incoming connections. Interface SocketMessengerConstants (Fig. 17.12) defines the port value as the constant SERVER_PORT to ensure that the server and the clients use the correct port number. Class DeitelMessengerServer implements interface SocketMessengerConstants to facilitate referencing the constants defined in that interface.

```java
import java.net.*;
import java.io.*;
import com.deitel.messenger.*;
import com.deitel.messenger.sockets.*;

public class DeitelMessengerServer implements MessageListener, SocketMessengerConstants {
    // start chat server
    public void startServer()
    {
        // create server and manage new clients
        try {
            // create ServerSocket for incoming connections
            ServerSocket serverSocket =
                new ServerSocket( SERVER_PORT, 100 );
            System.out.println( "Server listening on port " +
                SERVER_PORT + " ..." );

            // listen for clients constantly
            while ( true ) {
                // accept new client connection
                Socket clientSocket = serverSocket.accept();
                // create new ReceivingThread for receiving
                // messages from client
```
Lines 32–45 listen continuously for new client connections. Line 35 invokes method `accept` of class `ServerSocket` to wait for and accept a new client connection. Line 39 creates and starts a new `ReceivingThread` for the client. Class `ReceivingThread` (Fig. 17.14) is a `Thread` subclass that listens for new incoming messages from a particular client. The first argument to the `ReceivingThread` constructor is a `MessageListener` (Fig. 17.13), to which messages from the client should be delivered. Class `DeitelMessengerServer` implements interface `MessageListener` (line 15) and therefore can pass the `this` reference to the `ReceivingThread` constructor.
Method `messageReceived` (lines 57–66) is required by interface `MessageListener`. When each `ReceivingThread` receives a new message from a client, the `ReceivingThread` passes the message to a `MessageListener` through method `messageReceived`. Line 60 concatenates the `from String` with the separator `>>>` and the message body. Lines 64–65 create and start a new `MulticastSendingThread` to deliver `completeMessage` to all listening clients. Class `MulticastSendingThread` (Fig. 17.15) uses `multicast` as an efficient mechanism for sending one message to multiple clients. We discuss the details of multicasting shortly. Method `main` (lines 69–72) creates a new `DeitelMessengerServer` instance and starts the server.

Interface `SocketMessengerConstants` (Fig. 17.12) declares constants for use in the various classes that make up the Deitel messenger system. Classes can access these `static` constants either by referencing the constants through interface `SocketMessengerConstants` (e.g., `SocketMessengerConstants.SERVER_PORT`) or by implementing the interface and referencing the constants directly.

Line 9 defines the `String` constant `MULTICAST_ADDRESS`, which contains the address to which a `MulticastSendingThread` (Fig. 17.15) should send messages. This address is one of the addresses reserved for multicast, which we will describe soon. Line 12 defines the integer constant `MULTICAST_LISTENING_PORT`—the port on which clients should listen for new messages. Line 15 defines the integer constant `MULTICAST_SENDING_PORT`—the port to which a `MulticastSendingThread` should post new messages at the `MULTICAST_ADDRESS`. Line 18 defines the integer constant `SERVER_PORT`—the port on which `DeitelMessengerServer` listens for incoming client connections. Line 21 defines `String` constant `DISCONNECT_STRING`, which is the `String` that a client sends to `DeitelMessengerServer` when the user wishes to leave the chat room. Line 24 defines `String` constant `MESSAGE_SEPARATOR`, which separates the user name from the message body. Line 27 specifies the maximum message size in bytes.

```java
// SocketMessengerConstants.java
// SocketMessengerConstants defines constants for the port numbers
// and multicast address in DeitelMessenger
package com.deitel.messenger.sockets;

public interface SocketMessengerConstants {

    // address for multicast datagrams
    public static final String MULTICAST_ADDRESS = "230.0.0.1";

    // port for listening for multicast datagrams
    public static final int MULTICAST_LISTENING_PORT = 5555;

    // port for sending multicast datagrams
    public static final int MULTICAST_SENDING_PORT = 5554;

    // port for Socket connections to DeitelMessengerServer
    public static final int SERVER_PORT = 5000;
}
```

Fig. 17.12 `SocketMessengerConstants` declares constants for use throughout the `DeitelMessengerServer` and `DeitelMessenger` applications (part 1 of 2).
Many different classes in the Deitel messenger system receive messages. For example, `DeitelMessengerServer` receives messages from clients and delivers those messages to all chat room participants. As we will see, the user interface for each client also receives messages and displays those messages to the users. Each of the classes that receives messages implements interface `MessageListener` (Fig. 17.13). The interface declares method `messageReceived`, which allows an implementing class to receive chat messages. Method `messageReceived` takes two `String` arguments representing the name of the user who sent the message and the message body, respectively.

`DeitelMessengerServer` uses instances of class `ReceivingThread` (Fig. 17.14) to listen for new messages from each client. Class `ReceivingThread` extends class `Thread`. This enables `DeitelMessengerServer` to create an object of class `ReceivingThread` for each client, to handle messages from multiple clients at once. When `DeitelMessengerServer` receives a new client connection, `DeitelMessengerServer` creates a new `ReceivingThread` for the client, then continues listening for new client connections. The `ReceivingThread` listens for new messages from a single client and passes those messages back to the `DeitelMessengerServer` through method `messageReceived`.

```java
// MessageListener.java
// MessageListener is an interface for classes that wish to
// receive new chat messages.
package com.deitel.messenger;

public interface MessageListener {
    // receive new chat message
    public void messageReceived( String from, String message );
}
```

Fig. 17.12 `SocketMessengerConstants` declares constants for use throughout the `DeitelMessengerServer` and `DeitelMessenger` applications (part 2 of 2).

Fig. 17.13 `MessageListener` interface that defines method `messageReceived` for receiving new chat messages.
// ReceivingThread.java
// ReceivingThread is a Thread that listens for messages
// from a particular client and delivers messages to a
// MessageListener.
package com.deitel.messenger.sockets.server;

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

// Deitel packages
import com.deitel.messenger.*;
import com.deitel.messenger.sockets.*;

public class ReceivingThread extends Thread implements
SocketMessengerConstants {
    private BufferedReader input;
    private MessageListener messageListener;
    private boolean keepListening = true;

    // ReceivingThread constructor
    public ReceivingThread( MessageListener listener,
                           Socket clientSocket )
    {
        // invoke superclass constructor to name Thread
        super( "ReceivingThread: " + clientSocket );

        // set listener to which new messages should be sent
        messageListener = listener;

        // set timeout for reading from clientSocket and create
        // BufferedReader for reading incoming messages
        try {
            clientSocket.setSoTimeout( 5000 );
            input = new BufferedReader( new InputStreamReader( clientSocket.getInputStream() ) );
        }
        catch ( IOException ioException ) {
            ioException.printStackTrace();
        }
    }

    // handle exception creating BufferedReader
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }

    } // end ReceivingThread constructor

    // listen for new messages and deliver them to MessageListener
    public void run()
    {
        String message;
1032 Networking
Chapter 17

while ( keepListening ) {
    try {
        message = input.readLine();
    }
    catch ( InterruptedIOException interruptedIOException ) {
        continue;
    }
    catch ( IOException ioException ) {
        ioException.printStackTrace();
        break;
    }
    if ( message != null ) {
        StringTokenizer tokenizer = new StringTokenizer( message, MESSAGE_SEPARATOR );
        if ( tokenizer.countTokens() == 2 ) {
            messageListener.messageReceived( tokenizer.nextToken(), tokenizer.nextToken() );
        } else {
            if ( message.equalsIgnoreCase( MESSAGE_SEPARATOR + DISCONNECT_STRING ) ) {
                stopListening();
            }
        }
    }
}  // end if
} // end while

Fig. 17.14 ReceivingThread for listening for new messages from DeitelMessengerServer clients in separate Threads (part 2 of 3).
The ReceivingThread constructor (lines 24–47) takes as its first argument a MessageListener. The ReceivingThread will deliver new messages to this MessageListener by invoking method messageReceived of interface MessageListener. The Socket argument to the ReceivingThread constructor is the connection to a particular client. Line 28 invokes the Thread constructor to provide a unique name for each ReceivingThread instance. Naming the ReceivingThread this way can be useful when debugging the application. Line 31 sets the MessageListener to which the ReceivingThread should deliver new messages. Line 36 invokes method setTimeout of class Socket with an integer argument of 5000 milliseconds. Reading data from a Socket is a blocking call—the current thread is put in the blocked state (Fig. 15.1) while the thread waits for the read operation to complete. Method setTimeout specifies that, if no data is received in the given number of milliseconds, the Socket should issue an InterruptedException, which the current thread can catch, then continue executing. This technique prevents the current thread from deadlocking if no more data is available from the Socket. Lines 38–39 create a new BufferedReader for the clientSocket’s InputStream. The ReceivingThread uses this BufferedReader to read new messages from the client.

Method run (lines 50–116) listens continuously for new messages from the client. Lines 55–104 loop as long as the boolean variable keepListening is true. Line 59 invokes BufferedReader method readLine to read a line of data from the client. If more than 5000 milliseconds pass without reading any data, method readLine throws an InterruptedException, which indicates that the timeout set on line 36 has expired. Line 66 uses keyword continue to go to the next iteration of the while loop to continue listening for messages. Lines 70–73 catch an IOException, which indicates a more severe problem from method readLine. Line 71 prints a stack trace to aid in debugging the application, and line 72 uses keyword break to terminate the while loop.

```java
// close BufferedReader (also closes Socket)
try {
    input.close();
}

// handle exception closing BufferedReader
catch (IOException ioException) {
    ioException.printStackTrace();
}

// stop listening for incoming messages
public void stopListening() {
    keepListening = false;
}
```

Fig. 17.14 ReceivingThread for listening for new messages from DeitelMessengerServer clients in separate Threads (part 3 of 3).
When a client sends a message to the server, the client separates the user’s name from
the message body with the String MESSAGE_SEPARATOR. If there are no exceptions
thrown when reading data from the client and the message is not null (line 76), lines 80–
81 create a new StringTokenizer. This StringTokenizer separates each mes-
sage into two tokens delimited by MESSAGE_SEPARATOR. The first token is the sender’s
user name; the second token is the message. Line 85 checks for the proper number of
tokens, and lines 88–90 invoke method messageReceived of interface Message-
Listener to deliver the new message to the registered MessageListener. If the
StringTokenizer does not produce two tokens, lines 96–97 check the message to see
whether it matches the constant DISCONNECT_STRING, which would indicate that the
user wishes to leave the chat room. If the Strings match, line 99 invokes Receiving-
Thread method stopListening to terminate the ReceivingThread.

Method stopListening (lines 119–122) sets boolean variable keepListening
to false. This causes the while loop condition on line 55 to fail and causes the
ReceivingThread to close the client Socket (line 108). Then, method run returns,
which terminates the ReceivingThread’s execution.

MulticastSendingThread (Fig. 17.15) delivers DatagramPacket s containing
chat messages to a group of clients. Multicast is an efficient way to send data to many clients
without the overhead of broadcasting that data to every host on the Internet. To understand
multicast, let us look at a real-world analogy—the relationship between a magazine publisher
and that magazine’s subscribers. The magazine publisher produces a magazine and provides
the magazine to a distributor. Customers interested in that magazine obtain a subscription and
begin receiving the magazine in the mail from the distributor. This communication is quite
different from a television broadcast. When a television station produces a television pro-
gram, the station broadcasts that television show throughout a geographical region or perhaps
throughout the world by using satellites. Broadcasting a television show for 10,000 viewers
is no more expensive to the television station than broadcasting a television show for 100
viewers—the radio signal carrying the broadcast reaches a wide area. However, printing and
delivering a magazine to 10,000 readers would be much more expensive than printing and
delivering the magazine to 100 readers. Most magazine publishers could not stay in business
if they had to broadcast their magazines to everyone, so magazine publishers multicast their
magazines to a group of subscribers instead.

Using multicast, an application can “publish” DatagramPacket s to be delivered to
other applications—the “subscribers.” An application multicasts DatagramPacket s by
sending the DatagramPacket s to a multicast address, which is an IP address reserved
for multicast in the range from 224.0.0.0 to 239.255.255.255. Clients that wish to
receive these DatagramPacket s can connect to the appropriate multicast address to join
the group of subscribers—the multicast group. When an application sends a Datagram-
Packet to the multicast address, each client in the multicast group receives the Data-
gramPacket. Multicast DatagramPacket s, like unicast DatagramPacket s
(Fig. 17.7), are not reliable—packets are not guaranteed to reach any destination. Also, the
order in which the particular clients receive the datagrams is not guaranteed.

Class MulticastSendingThread extends class Thread to enable Deitel-
MessengerServer to send multicast messages in a separate thread. Each time
DeitelMessengerServer needs to multicast a message, the server creates a new
MulticastSendingThread with the contents of the message and starts the thread.
The `MulticastSendingThread` constructor (lines 20–26) takes as an argument an array of `bytes` containing the message.

```java
public class MulticastSendingThread extends Thread implements SocketMessengerConstants {
    private byte[] messageBytes;

    public MulticastSendingThread( byte[] bytes ) {
        super( "MulticastSendingThread" );
        messageBytes = bytes;
    }

    public void run() {
        try {
            DatagramSocket socket =
                new DatagramSocket( MULTICAST_SENDING_PORT );

            InetAddress group = InetAddress.getByName( MULTICAST_ADDRESS );

            DatagramPacket packet = new DatagramPacket( messageBytes, messageBytes.length, group,
                MULTICAST_LISTENING_PORT );

            socket.send( packet );
        }
    }
}
```

**Fig. 17.15** `MulticastSendingThread` for delivering outgoing messages to a multicast group via `DatagramPackets`. 
Method `run` (lines 29–57) delivers the message to the multicast address. Lines 35–36 create a new `DatagramSocket`. Recall from the packet-networking example that we used `DatagramSocket`s to send `unicast DatagramPacket`s—packets sent from one host directly to another host. Delivering `DatagramPacket`s by using multicast is exactly the same, except the address to which the `DatagramPacket`s are sent is a multicast address in the range from `224.0.0.0` to `239.255.255.255`. Lines 39–40 create an `InetAddress` object for the multicast address, which is defined as a constant in interface `SocketMessengerConstants`. Lines 43–45 create the `DatagramPacket` containing the message. The first argument to the `DatagramPacket` constructor is the `byte` array containing the message. The second argument is the length of the `byte` array. The third argument specifies the `InetAddress` to which the packet should be sent, and the last argument specifies the port number through which the packet should be delivered to the multicast address. Line 48 invokes method `send` of class `DatagramSocket` to send the `DatagramPacket`. When the `DatagramPacket` is delivered to the multicast address, all clients listening to that multicast address on the proper port receive the `DatagramPacket`. Line 49 closes the `DatagramSocket`, and the run method returns, terminating the `MulticastSendingThread`.

17.10.2 DeitelMessenger Client and Supporting Classes

The client for the `DeitelMessengerServer` consists of several pieces. A class that implements interface `MessageManager` (Fig. 17.16) manages communication with the server. A `Thread` subclass listens for messages at `DeitelMessengerServer`’s multicast address. Another `Thread` subclass sends messages from the client to `DeitelMessengerServer`. A `JFrame` subclass provides a GUI for the client.

Interface `MessageManager` (Fig. 17.16) defines methods for managing communication with `DeitelMessengerServer`. We define this interface to abstract the base functionality a client needs to interact with a chat server from the underlying communication mechanism needed to communicate with that chat server. This abstraction enables us to provide `MessageManager` implementations that use other network protocols to implement the communication details. For example, if we want to connect to a different chat server that does not use multicast `DatagramPacket`s, we could implement the `MessageManager` interface with the appropriate network protocols for this alternate messaging server. We would not need to modify any other code in the client, because the
other components of the client refer only to interface `MessageManager`, and not to some particular `MessageManager` implementation. Likewise, the `MessageManager` interface methods refer to other components of the client only through interface `MessageListener`. Therefore, other components of the client can change without requiring changes in the `MessageManager` or its implementations. Method `connect` (line 10) connects `MessageManager` to `DeitelMessengerServer` and routes incoming messages to the appropriate `MessageListener`. Method `disconnect` (line 14) disconnects the `MessageManager` from the `DeitelMessengerServer` and stops delivering messages to the given `MessageListener`. Method `sendMessage` (line 17) sends a new message to `DeitelMessengerServer`.

Class `SocketMessageManager` (Fig. 17.17) implements interface `MessageManager` (line 16), using `Sockets` and `MulticastSockets` to communicate with `DeitelMessengerServer` and receive incoming messages. Line 20 declares the `Socket` that `SocketMessageManager` uses to connect and send messages to `DeitelMessengerServer`. Line 26 declares the `PacketReceivingThread` (Fig. 17.19) that listens for new incoming messages. The `boolean` flag `connected` (line 29) indicates whether the `SocketMessageManager` is connected to `DeitelMessengerServer`.

```java
// 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 );

    // disconnect from message server and stop routing
    // incoming messages to given MessageListener
    public void disconnect( MessageListener listener );

    // send message to message server
    public void sendMessage( String from, String message );
}

Fig. 17.16 MessageManager interface that defines methods for communicating with a DeitelMessengerServer.
```

```java
// SocketMessageManager.java
// SocketMessageManager is a MessageManager implementation for
// communicating with a DeitelMessengerServer using Sockets
// and MulticastSockets.
package com.deitel.messenger.sockets.client;

Fig. 17.17 SocketMessageManager implementation of interface MessageManager for communicating via Sockets and multicast DatagramPackets (part 1 of 4).
```
import java.util.*;
import java.net.*;
import java.io.*;

import com.deitel.messenger.*;
import com.deitel.messenger.sockets.*;

public class SocketMessageManager implements MessageManager,
        SocketMessengerConstants {
    // Socket for outgoing messages
    private Socket clientSocket;

    // DeitelMessengerServer address
    private String serverAddress;

    // Thread for receiving multicast messages
    private PacketReceivingThread receivingThread;

    // flag indicating connection status
    private boolean connected = false;

    // SocketMessageManager constructor
    public SocketMessageManager( String address )
    {
        serverAddress = address;
    }

    // connect to server and send messages to given MessageListener
    public void connect( MessageListener listener )
    {
        // if already connected, return immediately
        if ( connected )
            return;

        // open Socket connection to DeitelMessengerServer
        try {
            clientSocket = new Socket(
                InetAddress.getByName( serverAddress ), SERVER_PORT );

            // create Thread for receiving incoming messages
            receivingThread = new PacketReceivingThread( listener );
            receivingThread.start();

            // update connected flag
            connected = true;
        }
    }

Fig. 17.17 SocketMessageManager implementation of interface MessageManager for communicating via Socket and multicast DatagramPacket (part 2 of 4).
// handle exception connecting to server

try {
    // notify server that client is disconnecting
    Thread disconnectThread = new SendingThread(clientSocket, "", DISCONNECT_STRING);
    disconnectThread.start();

    // wait 10 seconds for disconnect message to be sent
    disconnectThread.join(10000);

    // stop receivingThread and remove given MessageListener
    receivingThread.stopListening();

    // close outgoing Socket
    clientSocket.close();
} // end try

// handle exception disconnecting from server

catch ( IOException ioException ) {
    ioException.printStackTrace();
}

// handle exception joining disconnectThread

catch ( InterruptedException interruptedException ) {
    interruptedException.printStackTrace();
}

// update connected flag

connected = false;

} // end method disconnect

// send message to server

// SocketMessageManager implementation of interface MessageManager for communicating via Sockets and multicast DatagramPackets (part 3 of 4).
The `SocketMessageManager` constructor (lines 32–35) receives the address of the `DeitelMessengerServer` to which `SocketMessageManager` should connect. Method `connect` (lines 38–63) connects `SocketMessageManager` to `DeitelMessengerServer`. If `SocketMessageManager` was connected previously, line 42 returns from method `connect`. Lines 46–47 create a new `Socket` to communicate with `DeitelMessengerServer`. Line 47 creates an `InetAddress` object for the server’s address and uses the constant `SERVER_PORT` to specify the port on which the client should connect. Line 50 creates a new `PacketReceivingThread`, which listens for incoming multicast messages from `DeitelMessengerServer`. Line 51 starts `PacketReceivingThread`. Line 54 updates boolean variable `connected` to indicate that `SocketMessageManager` is connected to the server.

Method `disconnect` (lines 66–104) terminates the `SocketMessageManager`’s connection to `DeitelMessengerServer`. If `SocketMessageManager` is not connected, line 70 returns from method `disconnect`. Lines 76–77 create a new `SendingThread` (Fig. 17.18) to send `DISCONNECT_STRING` to `DeitelMessengerServer`. Class `SendingThread` delivers a message to `DeitelMessengerServer` over the `SocketMessageManager`’s `Socket` connection. Line 78 starts the `SendingThread` to deliver the message. Line 81 invokes `SendingThread` method `join` (inherited from `Thread`) to wait for the disconnect message to be delivered. The integer argument `10000` specifies that the current thread should wait only 10 seconds to `join` the `SendingThread` before continuing. Once the disconnect message has been delivered, line 84 invokes method `stopListening` of class `PacketReceivingThread` to stop receiving incoming chat messages. Line 87 closes the `Socket` connection to `DeitelMessengerServer`.

Method `sendMessage` (lines 107–115) sends an outgoing message to `DeitelMessengerServer`. If `SocketMessageManager` is not connected, line 111 returns from method `sendMessage`. Line 114 creates and starts a new `SendingThread` instance (Fig. 17.18) to deliver the new message in a separate thread of execution.

Class `SendingThread` (Fig. 17.18) extends class `Thread` to deliver outgoing messages to the `DeitelMessengerServer` in a separate thread of execution. `SendingThread`’s constructor (lines 21–31) takes as arguments the `Socket` over which to send the message, the `userName` from whom the message came and the `message` body. Line 30 concatenates `userName`, `MESSAGE_SEPARATOR` and `message` to build `messageToSend`. Constant `MESSAGE_SEPARATOR` enables the message recipient to parse the message into two parts—the sending user’s name and the message body—by using a `StringTokenizer`.

```java
// if not connected, return immediately
if ( !connected )
    return;

// create and start new SendingThread to deliver message
new SendingThread( clientSocket, from, message).start();
```
// SendingThread.java
// SendingThread sends a message to the chat server in a
// separate Thread.
package com.deitel.messenger.sockets.client;

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

// Deitel packages
import com.deitel.messenger.sockets.*;

public class SendingThread extends Thread
    implements SocketMessengerConstants {

    // Socket over which to send message
    private Socket clientSocket;
    private String messageToSend;

    // SendingThread constructor
    public SendingThread( Socket socket, String userName,
        String message ) {
        // invoke superclass constructor to name Thread
        super( "SendingThread: " + socket );

        clientSocket = socket;

        // build the message to be sent
        messageToSend = userName + MESSAGE_SEPARATOR + message;
    }

    // send message and exit Thread
    public void run() {
        // send message and flush PrintWriter
        try {
            PrintWriter writer =
                new PrintWriter( clientSocket.getOutputStream() );
            writer.println( messageToSend );
            writer.flush();
        } catch ( IOException ioException ) {
            ioException.printStackTrace();
        }
    }
}

Fig. 17.18 SendingThread for delivering outgoing messages to DeitelMessengerServer.
Method run (lines 34–49) delivers the complete message to DeitelMessengerServer, using the Socket provided to the SendingThread constructor. Lines 38–39 create a new PrintWriter for the clientSocket’s OutputStream. Line 40 invokes method println of class PrintWriter to send the message. Line 41 invokes method flush of class PrintWriter to ensure that the message is sent immediately. Note that class SendingThread does not close the clientSocket. Class SocketMessageManager uses a new instance of class SendingThread for each message the client sends, so the clientSocket must remain open until the user disconnects from DeitelMessengerServer.

Class PacketReceivingThread extends class Thread to enable SocketMessageManager to listen for incoming messages in a separate thread of execution. Line 19 declares the MessageListener to which PacketReceivingThread will deliver incoming messages. Line 22 declares a MulticastSocket, which enables PacketReceivingThread to receive multicast DatagramPackets. Line 25 declares an InetAddress reference for the multicast address to which DeitelMessengerServer posts new chat messages. The MulticastSocket connects to this InetAddress to listen for incoming chat messages.

```java
// PacketReceivingThread.java
// PacketReceivingThread listens for DatagramPackets containing messages from a DeitelMessengerServer.
package com.deitel.messenger.sockets.client;

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

// Deitel packages
import com.deitel.messenger.*;
import com.deitel.messenger.sockets.*;

public class PacketReceivingThread extends Thread implements SocketMessengerConstants {
    // MessageListener to whom messages should be delivered
    private MessageListener messageListener;

    // MulticastSocket for receiving broadcast messages
    private MulticastSocket multicastSocket;

    // InetAddress of group for messages
    private InetAddress multicastGroup;

    // flag for terminating PacketReceivingThread
    private boolean keepListening = true;
}
```

Fig. 17.19 PacketReceivingThread for listening for new multicast messages from DeitelMessengerServer in a separate Thread (part 1 of 4).
// PacketReceivingThread constructor
public PacketReceivingThread( MessageListener listener )
{
    // invoke superclass constructor to name Thread
    super( "PacketReceivingThread" );

    // set MessageListener
    messageListener = listener;

    // connect MulticastSocket to multicast address and port
    try {
        multicastSocket =
            new MulticastSocket( MULTICAST_LISTENING_PORT );

        multicastGroup =
            InetAddress.getByName( MULTICAST_ADDRESS );

        // join multicast group to receive messages
        multicastSocket.joinGroup( multicastGroup );

        // set 5 second time-out when waiting for new packets
        multicastSocket.setSoTimeout( 5000 );
    }

    // handle exception connecting to multicast address
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }

} // end PacketReceivingThread constructor

// listen for messages from multicast group
public void run()
{
    // listen for messages until stopped
    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 ) {

            Fig. 17.19 PacketReceivingThread for listening for new multicast messages from DeitelMessengerServer in a separate Thread (part 2 of 4).
// continue to next iteration to keep listening
continue;

// handle exception reading packet from multicast group
} catch ( IOException ioException ) {
  ioException.printStackTrace();
  break;
}

// put message data in a String
String message = new String( packet.getData() );

// ensure non-null message
if ( message != null ) {
  // trim extra whitespace from end of message
  message = message.trim();
  // tokenize message to retrieve user name
  // and message body
  StringTokenizer tokenizer =
    new StringTokenizer( message, MESSAGE_SEPARATOR );

  // ignore messages that do not contain a user
  // name and message body
  if ( tokenizer.countTokens() == 2 ) {
    // send message to MessageListener
    messageListener.messageReceived(
      tokenizer.nextToken(), // user name
      tokenizer.nextToken() ); // message body
  }
}

} // end if

} // end while

// leave multicast group and close MulticastSocket
try {
  multicastSocket.leaveGroup( multicastGroup );
  multicastSocket.close();
}

// handle exception reading packet from multicast group
} catch ( IOException ioException ) {
  ioException.printStackTrace();
}

} // end method run

Fig. 17.19 PacketReceivingThread for listening for new multicast messages
from DeitelMessengerServer in a separate Thread (part 3 of 4).
The `PacketReceivingThread` constructor (lines 31–59) takes as an argument the `MessageListener` to which the `PacketReceivingThread` should deliver incoming messages. Recall that interface `MessageListener` defines a single method `messageReceived`. When the `PacketReceivingThread` receives a new chat message over the `MulticastSocket`, `PacketReceivingThread` invokes method `messageReceived` to deliver the new message to the `MessageListener`.

Lines 41–42 create a new `MulticastSocket` and pass to the `MulticastSocket` constructor the constant `MULTICAST_LISTENING_PORT` from interface `SocketMessengerConstants`. This argument specifies the port on which the `MulticastSocket` should listen for incoming chat messages. Lines 44–45 create an `InetAddress` object for the `MULTICAST_ADDRESS`, to which `DeitelMessengerServer` multicasts new chat messages. Line 48 invokes method `joinGroup` of class `MulticastSocket` to register the `MulticastSocket` to receive messages sent to `MULTICAST_ADDRESS`. Line 51 invokes `MulticastSocket` method `setSoTime-out` to specify that, if no data is received in 5000 milliseconds, the `MulticastSocket` should issue an `InterruptedIOException`, which the current thread can catch, then continue executing. This approach prevents `PacketReceivingThread` from deadlocking when waiting for incoming data. Also, if the `MulticastSocket` did not ever time out, the while loop would not be able to check the `keepListening` variable and would therefore prevent `PacketReceivingThread` from stopping if `keepListening` were set to `false`.

Method `run` (lines 62–131) listens for incoming multicast messages. Line 68 creates a `byte` array in which to store the incoming `DatagramPacket` data. Lines 71–72 create a `DatagramPacket` to store the incoming message. Line 76 invokes method `receive` of class `MulticastSocket` with the `DatagramPacket` packet as an argument. This is a blocking call that reads an incoming packet from the multicast address. If 5000 milliseconds pass without receipt of a packet, method `receive` throws an `InterruptedIOException`, because we previously set a 5000 millisecond time-out (line 51). Line 83 uses keyword `continue` to proceed to the next while loop iteration to continue listening for incoming messages. For other `IOExceptions`, line 89 breaks the while loop to terminate the `PacketReceivingThread`.

Line 93 invokes method `getData` of class `DatagramPacket` to retrieve the message data. Line 99 invokes method `trim` of class `String` to remove extra whitespace from the end of the message. Recall that `DatagramPacket`s are a fixed size—512 bytes in this example—so, if the message is shorter than 512 bytes, there will be extra whitespace after the message. Lines 103–104 create a `StringTokenizer` to separate the message body from the name of the user who sent the message. Line 108 checks for the correct number of tokens. Lines 111–113 invoke method `messageReceived` of inter-
face `MessageListener` to deliver the incoming message to the `PacketReceivingThread`'s `MessageListener`.

If the program invokes method `stopListening` (lines 134–138), the `while` loop in method `run` (lines 62–118) terminates. Line 122 invokes method `leaveGroup` of class `MulticastSocket` to stop receiving messages from the multicast address. Line 123 invokes method `close` of class `MulticastSocket` to close the `MulticastSocket`. `PacketReceivingThread` then terminates when method `run` returns.

Class `ClientGUI` (Fig. 17.20) extends class `JFrame` to create a GUI for a user to send and receive chat messages. The GUI consists of a `JTextArea` for displaying incoming messages (line 22), a `JTextArea` for entering new messages (line 23), `JButtons` and `JMenuItem`s for connecting to and disconnecting from the server (lines 26–29) and a `JButton` for sending messages (line 32). The GUI also contains a `JLabel` that displays whether the client is connected or disconnected.

```java
package com.deitel.messenger;

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

public class ClientGUI extends JFrame {

    // JMenu for connecting/disconnecting server
    private JMenu serverMenu;

    // JTextAreas for displaying and inputting messages
    private JTextArea messageArea;
    private JTextArea inputArea;

    // JButtons and JMenuItem for connecting and disconnecting
    private JButton connectButton;
    private JMenuItem connectMenuItem;
    private JButton disconnectButton;
    private JMenuItem disconnectMenuItem;

    // JButton for sending messages
    private JButton sendButton;

    // JLabel for displaying connection status
    private JLabel statusBar;
```

Fig. 17.20 `ClientGUI` subclass of `JFrame` for presenting a GUI for viewing and sending chat messages (part 1 of 6).
Chapter 17  Networking  1047

// userName to add to outgoing messages
private String userName;

// MessageManager for communicating with server
private MessageManager messageManager;

// MessageListener for receiving incoming messages
private MessageListener messageListener;

// ClientGUI constructor
public ClientGUI( MessageManager manager )
{
    super( "Deitel Messenger" );
    // set the MessageManager
    messageManager = manager;
    // create MyMessageListener for receiving messages
    messageListener = new MyMessageListener();

    // create File JMenu
    serverMenu = new JMenu( "Server" );
    serverMenu.setMnemonic( 'S' );
    JMenuBar menuBar = new JMenuBar();
    menuBar.add( serverMenu );
    setJMenuBar( menuBar );

    // create ImageIcon for connect buttons
    Icon connectIcon = new ImageIcon(
        getClass().getResource( "images/Connect.gif" ) );

    // create JButton and JMenuItem
    connectButton = new JButton( "Connect", connectIcon );
    connectMenuIcon = new JMenuItem( "Connect", connectIcon );
    connectMenuIcon.setMnemonic( 'C' );

    // create ConnectListener for connect buttons
    ActionListener connectListener = new ConnectListener();
    connectButton.addActionListener( connectListener );
    connectMenuIcon.addActionListener( connectListener );

    // create ImageIcon for disconnect buttons
    Icon disconnectIcon = new ImageIcon(
        getClass().getResource( "images/Disconnect.gif" ) );

    // create JButton and JMenuItem
    disconnectButton = new JButton( "Disconnect", disconnectIcon );
    disconnectMenuIcon = new JMenuItem( "Disconnect", disconnectIcon );
    disconnectMenuIcon.setMnemonic( 'D' );

Fig. 17.20 ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages (part 2 of 6).
// disable disconnect buttons
disconnectButton.setEnabled( false );
disconnectMenuItem.setEnabled( false );

// create DisconnectListener for disconnect buttons
ActionListener disconnectListener =
    new DisconnectListener();
disconnectButton.addActionListener( disconnectListener );
disconnectMenuItem.addActionListener( disconnectListener );

// add connect and disconnect JMenuItem to fileMenu
serverMenu.add( connectMenuItem );
serverMenu.add( disconnectMenuItem );

// add connect and disconnect JButtons to buttonPanel
JPanel buttonPanel = new JPanel();
buttonPanel.add( connectButton );
buttonPanel.add( disconnectButton );

// create JTextArea for displaying messages
messageArea = new JTextArea();

// disable editing and wrap words at end of line
messageArea.setEditable( false );
messageArea.setWrapStyleWord( true );
messageArea.setLineWrap( true );

// put messageArea in JScrollPane to enable scrolling
JPanel messagePanel = new JPanel();
messagePanel.setLayout( new BorderLayout( 10, 10 ) );
messagePanel.add( new JScrollPane( messageArea ),
    BorderLayout.CENTER );

// create JTextArea for entering new messages
inputArea = new JTextArea( 4, 20 );
inputArea.setWrapStyleWord( true );
inputArea.setLineWrap( true );
inputArea.setEditable( false );

// create Icon for sendButton
Icon sendIcon = new ImageIcon(
    getClass().getResource( "images/Send.gif" ) );

// create sendButton and disable it
sendButton = new JButton( "Send", sendIcon );
sendButton.setEnabled( false );

// create ActionListener for sendButton
sendButton.addActionListener(
    new ActionListener() {

Fig. 17.20 ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages (part 3 of 6).
// send new message when user activates sendButton
public void actionPerformed( ActionEvent event )
{
    messageManager.sendMessage( userName,
        inputArea.getText());

    // clear inputArea
    inputArea.setText( "" );
}
} // end ActionListener

// lay out inputArea and sendButton in BoxLayout and
// add Box to messagePanel
Box box = new Box( BoxLayout.X_AXIS );
box.add( new JScrollPane( inputArea ) );
box.add( sendButton );
messagePanel.add( box, BorderLayout.SOUTH );

// create JLabel for statusBar with a recessed border
statusBar = new JLabel( "Not Connected" );
statusBar.setBorder(
    new BevelBorder( BevelBorder.LOWERED ) );

// lay out components in JFrame
Container container = getContentPane();
container.add( buttonPanel, BorderLayout.NORTH );
container.add( messagePanel, BorderLayout.CENTER );
container.add( statusBar, BorderLayout.SOUTH );

// add WindowListener to disconnect when user quits
addWindowListener ( new WindowAdapter () {

    // disconnect from server and exit application
    public void windowClosing ( WindowEvent event )
    {
        messageManager.disconnect( messageListener );
        System.exit( 0 );
    }
});

} // end ClientGUI constructor

// ConnectListener listens for user requests to connect to
// DeitelMessengerServer
private class ConnectListener implements ActionListener {

    // connect to server and enable/disable GUI components
    public void actionPerformed( ActionEvent event )
    {
// connect to server and route messages to
// messageListener
messageManager.connect( messageListener );

// prompt for userName
userName = JOptionPane.showInputDialog(
    ClientGUI.this, "Enter user name:" );

// clear messageArea
messageArea.setText( "" );

// update GUI components
connectButton.setEnabled( false );
connectMenuItem.setEnabled( false );
disconnectButton.setEnabled( true );
disconnectMenuItem.setEnabled( true );
sendButton.setEnabled( true );
inputArea.setEditable( true );
inputArea.requestFocus();
statusBar.setText( "Connected: " + userName );

} // end ConnectListener inner class

// DisconnectListener listens for user requests to disconnect
// from DeitelMessengerServer
private class DisconnectListener implements ActionListener {

    // disconnect from server and enable/disable GUI components
    public void actionPerformed( ActionEvent event )
    {
        // disconnect from server and stop routing messages
        // to messageListener
        messageManager.disconnect( messageListener );

        // update GUI components
        sendButton.setEnabled( false );
disconnectButton.setEnabled( false );
disconnectMenuItem.setEnabled( false );
inputArea.setEditable( false );
connectButton.setEnabled( true );
connectMenuItem.setEnabled( true );
statusBar.setText( "Not Connected" );

    } // end DisconnectListener inner class

// MyMessageListener listens for new messages from the
// MessageManager and displays the messages in messageArea
// using a MessageDisplayer.
private class MyMessageListener implements MessageListener {

Fig. 17.20 ClientGUI subclass of JFrame for presenting a GUI for viewing and sending chat messages (part 5 of 6).
ClientGUI uses a MessageManager (line 41) to handle all communication with the chat server. Recall that MessageManager is an interface and therefore allows ClientGUI to use any MessageManager implementation without the need to change any code in ClientGUI. Class ClientGUI also uses a MessageListener (line 44) to receive incoming messages from the MessageManager.

The ClientGUI constructor (lines 47–183) takes as an argument the MessageManager for communicating with DeitelMessengerServer. Line 52 sets the Cli-
entGUI’s MessageManager. Line 55 creates an instance of `MyMessageListener`, which implements interface `MessageListener`. Lines 58–62 create a `Server` menu that contains `JMenuItems` for connecting to and disconnecting from the chat server. Lines 65–66 create an `Icon` for `connectButton` and `connectMenuItem`.

Line 66 invokes method `getClass` (inherited from class `Object`) to retrieve the `Class` object that represents the `ClientGUI` class definition. Line 66 then invokes method `getResource` of class `Class` to load the connect image. The Java virtual machine loads class definitions into memory, using a `class loader`. Method `getResource` of `Class` uses the `Class` object’s class loader to specify the location of a resource, such as an image file. Specifying resource locations in this manner enables programs to avoid hard-coded or absolute paths, which can make programs more difficult to deploy. Using the techniques described here enables an applet or application to load files from locations that are relative to the location of the `.class` file for a given class.

Lines 69–70 create `connectButton` and `connectMenuItem`, each with the label "Connect" and the `Icon` `connectIcon`. Line 71 invokes method `setMnemonic` of class `JMenuItem` to set the mnemonic character for keyboard access to `connectMenuItem`. Line 74 creates an instance of `private` inner class `ConnectListener`, which implements interface `ActionListener` to handle `ActionEvents` from `connectButton` and `connectMenuItem`. Lines 75–76 add `connectListener` as an `ActionListener` for `connectButton` and `connectMenuItem`

Lines 79–80 create an `ImageIcon` for the `disconnectButton` and `disconnectMenuItem` components. Lines 83–86 create `disconnectButton` and `disconnectMenuItem`, each with the label "Disconnect" and the `Icon` `disconnectIcon`. Line 87 invokes method `setMnemonic` of class `JMenuItem` to enable keyboard access to `disconnectMenuItem`. Lines 90–91 invoke method `setEnabled` of class `JButton` and class `JMenuItem` with a `false` argument to disable `disconnectButton` and `disconnectMenuItem`. This prevents the user from attempting to disconnect from the server because the client is not yet connected. Lines 94–95 create an instance of `private` inner class `DisconnectListener`, which implements interface `ActionListener` to handle `ActionEvents` from `disconnectButton` and `disconnectMenuItem`. Lines 96–97 add `disconnectListener` as an `ActionListener` for the `disconnectButton` and `disconnectMenuItem` components.

Lines 100–101 add `connectMenuItem` and `disconnectMenuItem` to the `Server` `JMenu`. Lines 104–106 create a `JPanel` and add `connectButton` and `disconnectButton` to that `JPanel`. Line 109 creates the `JTextArea` `messageArea`, in which the client displays incoming messages. Line 112 invokes method `setEnabled` with a `false` argument, to disable editing of the text in `messageArea`. Lines 113–114 invoke methods `setWrapStyleWord` and `setLineWrap` of class `JTextArea` to enable word wrapping in `messageArea`. If a message is longer than the width of the `messageArea`, the `messageArea` will wrap the text after the last word that fits on each line, making longer messages easier to read. Lines 117–120 create a `JPanel` for the `messageArea` and add the `messageArea` to the `JPanel` in a `JScrollPane`. The `JScrollPane` adds scroll bars to the `messageArea` to enable the user to scroll through messages that exceed the size of `messageArea`

Line 123 creates the `inputArea JTextArea` for entering new messages. The arguments to the `JTextArea` constructor specify a four-line `JTextArea` that is twenty char-
acters wide. Lines 124–125 enable word and line wrapping, and line 126 disables editing the inputArea. When the client connects to the chat server, ConnectListener enables the inputArea to allow the user to type new messages.

Line 129 creates an ImageIcon for sendButton. Line 133 creates sendButton, which the user can click to send a message the user has typed. Line 134 disables sendButton; the ConnectListener enables the sendButton when the client connects to the chat server. Lines 137–150 add an ActionListener to sendButton. Lines 143–144 invoke method sendMessage of interface MessageManager with the userName and inputArea text as arguments. This statement sends the user’s name and whatever text the user entered in inputArea to DeitelMessengerServer as a new chat message. Line 147 invokes method setText of class JTextArea with an empty String argument to clear the inputArea for the next message.

Lines 154–157 use a BoxLayout to arrange the inputArea and sendButton. Line 155 places inputArea in a JScrollPane to enable scrolling of long messages. Line 157 adds the Box containing inputArea and sendButton to the SOUTH region of messagePanel. Lines 160–162 create the statusBar JLabel. This JLabel displays whether the client is connected to or disconnected from the chat server. Lines 161–162 invoke method setBorder of class JLabel and create a new BevelBorder of type BevelBorder.LOWERED. This border makes the JLabel appear recessed, as is common with status bars in many applications. Lines 165–168 lay out buttonPanel, messagePanel and statusBar in the ClientGUI JFrame.

Lines 171–181 add a WindowListener to the ClientGUI JFrame. Line 177 invokes method disconnect of interface MessageManager to disconnect from the chat server in case the user quits while still connected.

Inner class ConnectListener (lines 187–214) handles events from connectButton and connectMenuItem. Line 194 invokes method connect of class MessageManager to connect to the chat server. Line 194 passes as an argument to method connect the MessageListener to which new messages should be delivered. Lines 197–198 prompt the user for a user name, and line 201 clears the messageArea JTextArea. Lines 204–209 enable the components for disconnecting from the server and for sending messages and disable components for connecting to the server. Line 210 invokes method requestFocus of class JTextArea to place the text-input cursor in the inputArea so the user can begin typing a message more easily.

Inner class DisconnectListener (lines 218–237) handles events from disconnectButton and disconnectMenuItem. Line 225 invokes method disconnect of class MessageManager to disconnect from the chat server. Lines 228–234 disable the components for sending messages and the components for disconnecting then enable the components for connecting to the chat server.

Inner class MyMessageListener (lines 242–254) implements interface MessageListener to receive incoming messages from the MessageManager. When a new message is received, the program invokes method messageReceived (lines 242–252) with the user name of the sender and the message body. Lines 249–250 invoke static method invokeLater of class SwingUtilities with a new instance of MessageDisplayer to append the new message to messageArea. Recall, from Chapter 15, that Swing components should be accessed only from the event dispatching thread. Method messageReceived is invoked by the PacketReceivingThread in
class **SocketMessageManager** and therefore cannot append the message text to **messageArea** directly, as this would occur in **PacketReceivingThread**, not the event-dispatch thread.

Inner class **MessageDisplayer** (lines 260–285) implements interface **Runnable** to provide a thread-safe way to append text to the **messageArea JTextArea**. The **MessageDisplayer** constructor (lines 266–270) takes as arguments the user name and message to send. Method **run** (lines 273–283) appends the user name, "> " and **messageBody** to **messageArea**. Lines 281–282 invoke method **setCaretPosition** of class **JTextArea** to scroll **messageArea** to the bottom to display the most recently received message. Instances of class **MessageDisplayer** should execute only as part of the event-dispatching thread, to ensure thread-safe access to the **messageArea** Swing component.

Class **DeitelMessenger** (Fig. 17.21) launches the client for the **DeitelMessengerServer**. Lines 18–21 create a new **SocketMessageManager** to connect to the **DeitelMessengerServer** with the IP address specified as a command-line argument to the application. Lines 24–27 create a **ClientGUI** for the **MessageManager**, set the **ClientGUI** size and make the **ClientGUI** visible.

```java
// DeitelMessenger.java
// DeitelMessenger is a chat application that uses a ClientGUI
// and SocketMessageManager to communicate with
// DeitelMessengerServer.
package com.deitel.messenger.sockets.client;

// Deitel packages
import com.deitel.messenger.*;

public class DeitelMessenger {

    // execute application
    public static void main( String args[] ) {

        MessageManager messageManager;

        // create new DeitelMessenger
        if ( args.length == 0 )
            messageManager = new SocketMessageManager( "localhost" );
        else
            messageManager = new SocketMessageManager( args[ 0 ] );

        // create GUI for SocketMessageManager
        ClientGUI clientGUI = new ClientGUI( messageManager );
        clientGUI.setSize( 300, 400 );
        clientGUI.setResizable( false );
        clientGUI.setVisible( true );
    }
}
```

*Fig. 17.21* **DeitelMessenger** application for participating in a **DeitelMessengerServer** chat session (part 1 of 3).
Fig. 17.21 *DeitelMessenger* application for participating in a *DeitelMessengerServer* chat session (part 2 of 3).
The Deitel messenger case study is a significant application that uses many intermediate Java features, such as networking with Sockets, DatagramPackets and MulticastSockets, multithreading and Swing GUI. The case study also demonstrates good software engineering practices by separating interface from implementation, enabling developers to build MessageManagers for different network protocols and MessageListeners that provide different user interfaces. You should now be able to apply these techniques to your own, more complex, Java projects.

17.11 (Optional) Discovering Design Patterns: Design Patterns Used in Packages java.io and java.net

This section introduces those design patterns associated with the Java file, streams and networking packages.

17.11.1 Creational Design Patterns

We now continue our discussion of creational design patterns.

Abstract Factory

Like the Factory Method design pattern, the Abstract Factory design pattern allows a system to determine the subclass from which to instantiate an object at run time. Often, this subclass is unknown during development. However, Abstract Factory uses an object known as a factory that uses an interface to instantiate objects. A factory creates a product; in this case, that product is an object of a subclass determined at run time.

The Java socket library in package java.net uses the Abstract Factory design pattern. A socket describes a connection, or a stream of data, between two computers. Class Socket references an object of a SocketImpl subclass (Section 17.5). Class Socket also contains a static reference to an object implementing interface SocketImplFactory. The Socket constructor invokes method createSocketImpl of interface SocketFactory to create the SocketImpl object. The object that imple-
ments interface `SocketFactory` is the factory, and an object of a `SocketImpl` subclass is the product of that factory. The system cannot specify the `SocketImpl` subclass from which to instantiate until run-time, because the system has no knowledge of what type of `Socket` implementation is required (e.g., a socket configured to the local network’s security requirements). Method `createSocketImpl` decides the `SocketImpl` subclass from which to instantiate the object at run-time.

### 17.11.2 Structural Design Patterns

This section concludes our discussion of structural design patterns.

#### Decorator

Let us reexamine class `CreateSequentialFile` (Fig. 16.6). Lines 127–128 of this class allow an `ObjectOutputStream` object, which writes objects to a file, to gain the responsibilities of a `FileOutputStream` object, which provides methods for writing bytes to files. Class `CreateSequentialFile` appears to “chain” objects—a `FileOutputStream` object is the argument to the `ObjectOutputStream`’s constructor. The fact that the `ObjectOutputStream` object can gain the behavior of a `FileOutputStream` dynamically prevents the need for creating a separate class called `ObjectFileOutputStream`, which would implement the behaviors of both classes.

Lines 127–128 of class `CreateSequentialFile` show an example of the Decorator design pattern, which allows an object to gain additional responsibilities dynamically. Using this pattern, designers do not have to create separate, unnecessary classes to add responsibilities to objects of a given class.

Let us consider a more complex example to discover how the Decorator design pattern can simplify a system’s structure. Suppose we wanted to enhance the I/O-performance of the previous example by using a `BufferedOutputStream`. Using the Decorator design pattern, we would write

```java
output = new ObjectOutputStream(
    new BufferedOutputStream(
        new FileOutputStream( fileName ) ) );
```

We can chain objects in this manner, because `ObjectOutputStream`, `BufferedOutputStream` and `FileOutputStream` extend abstract superclass `OutputStream`, and each subclass constructor takes an `OutputStream` object as a parameter. If the stream objects in package `java.io` did not use the Decorator pattern (i.e., did not satisfy these two requirements), package `java.io` would have to provide classes `BufferedFileOutputStream`, `ObjectBufferedOutputStream`, `ObjectBufferedFileOutputStream` and `ObjectFileOutputStream`. Consider how many classes we would have to create if we chained even more stream objects without applying the Decorator pattern.

#### Facade

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. This principle is the foundation of the Facade design pattern, which allows an object—called a facade object—to provide a simple interface for the behaviors of a subsystem—an aggregate of objects that comprise collectively 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 access 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. The Facade design pattern reduces system complexity, because a client interacts with only one object (the facade). This 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.

In package java.net, an object of class URL is a facade object. This object contains a reference to an InetAddress object that specifies the host computer’s IP address. The URL facade object also references an object from class URLStreamHandler, which opens the URL connection. The client object that uses the URL facade object accesses the InetAddress object and the URLStreamHandler object through the facade object. However, the client object does not know how the objects behind the URL facade object accomplish their responsibilities.

17.11.3 Architectural Patterns

Design patterns allow developers to design specific parts of systems, such as abstracting object instantiations or aggregating classes into larger structures. Design patterns also promote loose coupling among objects. Architectural patterns promote loose coupling among subsystems. These patterns specify all subsystems in the system and how they interact with each other.\(^1\) We introduce the popular Model-View-Controller and Layers architectural patterns.

MVC

Consider the design of a simple text editor. In this program, the user inputs text from the keyboard and formats this text using the mouse. Our 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.

This program adheres to the Model-View-Controller (MVC) architectural pattern, which separates application data (contained in the model) from graphical presentation components (the view) and input-processing logic (the controller).\(^2\) Figure 17.22 shows the relationships between components in MVC.

---

2. Section 13.17 also discussed Model-View-Controller architecture and its relevance to the elevator simulation case study.
The controller implements logic for processing user inputs. The model contains application data, and the view presents the data stored in the model. When a user provides some input, the controller modifies the model with the given input. The model contains the application data. With regards to the text-editor example, the model might contain only the characters that make up the document. When the model changes, it notifies the view of the change so the view can update its presentation with the changed data. The view in a word processor might display characters using a particular font, with a particular size, etc.

MVC does not restrict an application to a single view and a single controller. In a more sophisticated program (e.g., a word processor), there might be two views of a document model. One view might display an outline of the document and the other might display the complete document. The word processor also might implement multiple controllers—one for handling keyboard input and another for handling mouse selections. If either controller makes a change in the model, both the outline view and the print-preview window will show the change immediately when the model notifies all views of changes.

Another key benefit to the MVC architectural pattern is that developers can modify each component individually without having to modify the other components. For example, developers could modify the view that displays the document outline, but the developers would not have to modify either the model or other views or controllers.

**Layers**

Consider the design in Fig. 17.23, which presents the basic structure of a three-tier application, in which each tier contains a unique system component.
The information tier (also called the “bottom tier”) maintains data for the application, typically storing the data in a database. The information tier for an online store may contain product information, such as descriptions, prices and quantities in stock and customer information, such as user names, billing addresses and credit-card numbers.

The middle tier acts as an intermediary between the information tier and the client tier. The middle tier processes client-tier requests, reads data from and writes data to the database. The middle tier then processes data from the information tier and presents the content to the client tier. This processing is the application’s business logic, which handles such tasks as retrieving data from the information tier, ensuring that data is reliable before updating the database and presenting data to the client tier. For example, the business logic associated with the middle tier for the online store can verify a customer’s credit card with the credit-card issuer before the warehouse ships the customer’s order. This business logic could then store (or retrieve) the credit information in the database and notify the client tier that the verification was successful.

The client tier (also called the “top tier”) is the application’s user interface, such as a standard Web browser. Users interact directly with the application through the user interface. The client tier interacts with the middle tier to make requests and retrieve data from the information tier. The client tier then displays data retrieved from the middle tier.

Figure 17.23 is an implementation of the Layers architectural pattern, which divides functionality into separate layers. Each layer contains a set of system responsibilities and depends on the services of only the next lower layer. In Fig. 17.23, each tier corresponds to a layer. This architectural pattern is useful, because a designer can modify one layer without having to modify the other layers. For example, a designer could modify the information tier in Fig. 17.23 to accommodate a particular database product, but the designer would not have to modify either the client tier or the middle tier.

17.11.4 Conclusion

In this “Discovering Design Patterns” section, we discussed how packages java.io and java.net take advantage of specific design patterns and how developers can integrate design patterns with networking/file applications in Java. We also introduced the Model-View-Controller and Layers architectural patterns, which both assign system functionality to separate subsystems. These patterns make designing a system easier for developers. In “Discovering Design Patterns” Section 21.12, we conclude our presentation of design patterns by discussing those design patterns used in package java.util.

SUMMARY

- Java provides stream sockets and datagram sockets. With stream sockets, a process establishes a connection to another process. While the connection is in place, data flows between the processes in continuous streams. Stream sockets are said to provide a connection-oriented service. The protocol used for transmission is the popular TCP (Transmission Control Protocol).

- With datagram sockets, individual packets of information are transmitted. This is not the right protocol for everyday users, because, unlike TCP, the protocol used, UDP—the User Datagram Protocol—is a connectionless service and does not guarantee that packets arrive in any particular way. In fact, packets can be lost, can be duplicated and can even arrive out of sequence. So, with UDP, significant extra programming is required on the user’s part to deal with these problems (if the user chooses to do so).
• The HTTP protocol (Hypertext Transfer Protocol) that forms the basis of the World Wide Web uses URIs (Uniform Resource Identifiers, also called URLs or Uniform Resource Locators) to locate data on the Internet. Common URIs represent files or directories and can represent complex tasks such as database lookups and Internet searches.
• Web browsers often restrict an applet so that it can communicate only with the machine from which it was originally downloaded.
• A **Hashtable** stores key/value pairs. A program uses a key to store and retrieve an associated value in the **Hashtable**. The **Hashtable** method **put** takes two arguments—a key and its associated value—and places the value in the **Hashtable** at a location determined by the key. The **Hashtable** method **get** takes one argument—a key—and retrieves the value (as an **Object** reference) associated with the key.
• A **Vector** is a dynamically resizable array of **Objects**. The **Vector** method **addElement** adds a new element to the end of the **Vector**.
• **Applet** method **getAppletContext** returns a reference to an **AppletContext** object that represents the applet’s environment (i.e., the browser in which the applet is executing).
• **AppletContext** method **showDocument** receives a **URL** object as an argument and passes it to the **AppletContext** (i.e., the browser), which displays the World Wide Web resource associated with that **URL**.
• A second version of **AppletContext** method **showDocument** enables an applet to specify the target frame in which Web resource should be displayed. Special target frames include _blank (display the content from the specified URI in a new Web browser window), _self (display the content from the specified URI in the same frame as the applet) and _top (the browser should remove the current frames, then display the content from the specified URI in the current window).
• **Component** method **setCursor** changes the mouse cursor when the cursor is positioned over a specific GUI component. The **Cursor** constructor receives an integer indicating the cursor type (such as **Cursor.WAIT_CURSOR** or **Cursor.DEFAULT_CURSOR**).
• **JEditorPane** method **setPage** downloads the document specified by its argument and displays it in the **JEditorPane**.
• Typically, an HTML document contains hyperlinks—text, images or GUI components that, when clicked, link to another document on the Web. If an HTML document is displayed in a **JEditorPane** and the user clicks a hyperlink, the **JEditorPane** generates a **HyperlinkEvent** (package **javax.swing.event**) and notifies all registered **HyperlinkListeners** (package **javax.swing.event**) of that event.
• **HyperlinkEvent** method **getEventType** determines the type of the **HyperlinkEvent**. The **HyperlinkEvent** class contains **public static** inner class **EventType**, which defines three hyperlink event types: **ACTIVATED** (user clicked a hyperlink), **ENTERED** (user moved the mouse over a hyperlink) and **EXITED** (user moved the mouse away from a hyperlink).
• **HyperlinkEvent** method **getURL** obtains the **URL** represented by the hyperlink.
• Stream-based connections are managed with **Socket** objects.
• A **ServerSocket** object establishes the port where a server waits for connections from clients. The second argument to the **ServerSocket** constructor specifies the number of clients that can wait for a connection and be processed by the server. If the queue of clients is full, client connections are refused. The **ServerSocket** method **accept** waits indefinitely (i.e., blocks) for a connection from a client and returns a **Socket** object when a connection is established.
• **Socket** method **getOutputStream** gets a reference to the **OutputStream** associated with a **Socket**. The **Socket** method **getInputStream** gets a reference to the **InputStream** associated with the **Socket**.
• When transmission over a Socket connection is complete, the server closes the connection by invoking the Socket’s close method.

• A Socket object connects a client to a server by specifying the server name and port number when creating the Socket object. A failed connection attempt throws an IOException.

• When the InputStream method read returns -1, the stream detects that the end-of-stream has been reached.

• An EOFException occurs when a ObjectInputStream attempts to read a value from a stream on which end-of-stream is detected.

• InetAddress method getByName returns an InetAddress object containing the host name of the computer for which the String host name or String Internet address is specified as an argument.

• InetAddress method getLocalHost returns an InetAddress object containing the host name of the local computer executing the program.

• The port at which a client connects to a server is sometimes called the handshake point.

• Connection-oriented transmission is like the telephone system—you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for the duration of your phone call, even when you are not talking.

• Connectionless transmission with datagrams is similar to mail carried via the postal service. A large message that will not fit in one envelope can be broken into separate message pieces that are placed in separate, sequentially numbered envelopes. Each of the letters is then mailed at once. The letters could arrive in order, out of order or not at all.

• DatagramPacket objects store packets of data for sending or store packets of data received by an application. DatagramSockets send and receive DatagramPackets.

• The DatagramSocket constructor that takes no arguments binds the application to a port chosen by the computer on which the program executes. The DatagramSocket constructor that takes an integer port number argument binds the application to the specified port. If a DatagramSocket constructor fails to bind the application to a port, a SocketException occurs.

• DatagramSocket method receive blocks (waits) until a packet arrives, then stores the packet in its argument.

• DatagramPacket method getAddress returns an InetAddress object containing information about the host computer from which the packet was sent.

• DatagramPacket method getPort returns an integer specifying the port number through which the host computer sent the DatagramPacket.

• DatagramPacket method getLength returns an integer representing the number of bytes of data in a DatagramPacket.

• DatagramPacket method getData returns a byte array containing the data in a DatagramPacket.

• The DatagramPacket constructor for a packet to be sent takes four arguments—the byte array to be sent, the number of bytes to be sent, the client address to which the packet will be sent and the port number where the client is waiting to receive packets.

• DatagramSocket method send sends a DatagramPacket out over the network.

• If an error occurs when receiving or sending a DatagramPacket, an IOException occurs.

• Reading data from a Socket is a blocking call—the current thread is put in the blocked state while the thread waits for the read operation to complete. Method setSoTimeout specifies that, if no data is received in the given number of milliseconds, the Socket should issue an Inter-
ruptedIOException, which the current thread can catch, then continue executing. This prevents the current thread from deadlocking if there is no more data available from the Socket.

- Multicast is an efficient way to send data to many clients without the overhead of broadcasting that data to every host on the Internet.
- Using multicast, an application can “publish” DatagramPackets to be delivered to other applications—the “subscribers.”
- An application multicasts DatagramPackets by sending the DatagramPackets to a multicast address—an IP address in the range from 224.0.0.0 to 239.255.255.255, reserved for multicast.
- Clients that wish to receive DatagramPackets can connect to the multicast address to join the multicast group that will receive the published DatagramPackets.
- Multicast DatagramPackets are not reliable—packets are not guaranteed to reach any destination. Also, the order in which clients receive the datagrams is not guaranteed.
- The MulticastSocket constructor takes as an argument the port to which the MulticastSocket should connect to receive incoming DatagramPackets. Method joinGroup of class MulticastSocket takes as an argument the InetAddress of the multicast group to join.
- Method receive of class MulticastSocket reads an incoming DatagramPacket from a multicast address.

**TERMINOLOGY**

- accept a connection
- accept method of ServerSocket class
- addElement method of class Vector
- AppletContext interface
- bind to a port
- BindException class
- client connects to a server
- client/server relationship
- client-side socket
- close a connection
- close method of class Socket
- collaborative computing
- computer networking
- connect to a port
- connect to a World Wide Web site
- ConnectException class
- connection
- connection request
- connectionless service
- connectionless transmission with datagrams
- connection-oriented service
- Cursor class
- Cursor.DEFAULT_CURSOR
- Cursor.WAIT_CURSOR
- datagram
- datagram socket
- DatagramPacket class
- DatagramSocket class
- deny a connection
- duplicated packets
- get method of class Hashtable
- getAddress method of DatagramPacket
- getApplicationContext method of class Applet
- getByName method of InetAddress
- getData method of class DatagramPacket
- getEventType method
- getInputStream method of class Socket
- getLength method of DatagramPacket
- getLocalHost method
- getLocalHost method of InetAddress
- getOutputStream method of class Socket
- getPort method of class DatagramPacket
- getPredefinedCursor method of Cursor
- getURL method of class HyperlinkEvent
- handshake point
- Hashtable class
- heterogeneous computer systems
- host
- Hyperlink.EventType class
- Hyperlink.EventType.ACTIVATED
- Hyperlink.EventType.ENTERED
- Hyperlink.EventType.EXITED
- HyperlinkEvent class
- HyperlinkListener interface
- hyperlinkUpdate method
- InetAddress class
- Internet
SELF-REVIEW EXERCISES

17.1 Fill in the blanks in each of the following statements:

a) Exception _______ occurs when an input/output error occurs when closing a socket.
b) Exception _______ occurs when a server address indicated by a client cannot be re-
resolved.
c) If a DatagramSocket constructor fails to set up a DatagramSocket properly, an exception of type _______ occurs.
d) Many of Java’s networking classes are contained in package _______.
e) Class _______ binds the application to a port for datagram transmission.
f) An object of class _______ contains an Internet address.
g) The two types of sockets we discussed in this chapter are _______ sockets and _______ sockets.
h) The acronym URL stands for _______. 
i) The acronym URI stands for _______. 
j) The key protocol that forms the basis of the World Wide Web is _______.
k) AppletContext method _______ receives a URL object as an argument and displays in a browser the World Wide Web resource associated with that URL.
l) InetAddress method getLocalHost returns an _______ object containing the local host name of the computer on which the program is executing.
m) Method _______ of class MulticastSocket subscribes the MulticastSocket to a multicast group.

Internet address
InterruptedIOException class
IOException class
Java Security API
javadoc package
JEditorPane class
joinGroup method of MulticastSocket
lost packets
MalformedURLException class
multicast
multicast address
multicast group
MulticastSocket class
multithreaded server
network programming
networking
open a socket
out-of-sequence packets
packet
packet length
port
port number on a server
put method of class Hashtable
read from a socket
receive method of class DatagramSocket
register an available port number
send method of class DatagramSocket
server
server-side socket
ServerSocket class
setCursor method of class Component
setPage method of class JEditorPane
setSoTimeout method of class Socket
showDocument method of AppletContext
socket
Socket class
socket-based communications
SocketException
stream socket
TCP (Transmission Control Protocol)
UDP (User Datagram Protocol)
UnknownHostException
URI (Uniform Resource Identifier)
URL (Uniform Resource Locator)
URL class
Vector class
wait for a connection
wait for a packet
Web browser
Web server
World Wide Web
n) The URL constructor determines whether the String passed as an argument represents a valid Uniform Resource Identifier. If so, the URL object is initialized to contain the URI; otherwise, an exception of type __________ occurs.

17.2 State whether each of the following is true or false. If false, explain why.
a) An application that uses multicast broadcasts DatagramPackets to every host on the Internet.
b) UDP is a connection-oriented protocol.
c) With stream sockets a process establishes a connection to another process.
d) A server waits at a port for connections from a client.
e) Datagram packet transmission over a network is reliable—packets are guaranteed to arrive in sequence.
f) For security reasons, many Web browsers such as Netscape Communicator allow Java applets to do file processing only on the machines on which they execute.
g) Web browsers often restrict an applet so that it can only communicate with the machine from which it was originally downloaded.
h) IP addresses in the range from 224.0.0.0 to 239.255.255.255 are reserved for multicast.

ANSWERS TO SELF-REVIEW EXERCISES

17.1 a) IOException.  b) UnknownHostException.  c) SocketException.  d) java.net.  e) DatagramSocket.  f) InetAddress.  g) stream, datagram.  h) Uniform Resource Locator.  i) Universal Resource Identifier.  j) http.  k) showDocument.  l) InetAddress.  m) joinGroup.  n) MalformedURLException.

17.2 a) False; multicast sends DatagramPackets only to hosts that have joined the multicast group.  b) False; UDP is a connectionless protocol and TCP is a connection-oriented protocol.  c) True.  d) True.  e) False; packets could be lost and packets can arrive out of order.  f) False; most browsers prevent applets from doing file processing on the client machine.  g) True.  h) True.

EXERCISES

17.3 Distinguish between connection-oriented and connectionless network services.
17.4 How does a client determine the host name of the client computer?
17.5 Under what circumstances would a SocketException be thrown?
17.6 How can a client get a line of text from a server?
17.7 Describe how a client connects to a server.
17.8 Describe how a server sends data to a client.
17.9 Describe how to prepare a server to receive a stream-based connection request from a single client.
17.10 Describe how to prepare a server to receive connection requests from multiple clients if each client that connects should be processed in parallel with all other connected clients.
17.11 How does a server listen for connections at a port?
17.12 What determines how many connect requests from clients can wait in a queue to connect to a server?
17.13 As described in the text, what reasons might cause a server to refuse a connection request from a client?
17.14 Use a socket connection to allow a client to specify a file name and have the server send the contents of the file or indicate that the file does not exist.

17.15 Modify Exercise 17.14 to allow the client to modify the contents of the file and send the file back to the server for storage. The user can edit the file in a JTextArea, then click a save changes button to send the file back to the server.

17.16 Modify program of Fig. 17.2 to allow users to add their own sites to the list and remove sites from the list.

17.17 Multithreaded servers are quite popular today, especially because of the increasing use of multiprocessing servers. Modify the simple server application presented in Section 17.6 to be a multithreaded server. Then use several client applications and have each of them connect to the server simultaneously. Use a Vector to store the client threads. Vector provides several methods of use in this exercise. Method size determines the number of elements in a Vector. Method elementAt returns the element in the specified location (as an Object reference). Method add places a new element at the end of the Vector. Method remove deletes its argument from the Vector. Method lastElement returns an Object reference to the last object you inserted in the Vector.

17.18 In the text, we presented a tic-tac-toe program controlled by a multithreaded server. Develop a checkers program modeled after the tic-tac-toe program. The two users should alternate making moves. Your program should mediate the players’ moves, determining whose turn it is and allowing only valid moves. The players themselves will determine when the game is over.

17.19 Develop a chess-playing program modeled after the checkers program in the Exercise 17.18.

17.20 Develop a Blackjack card game program in which the server application deals cards to each of the client applets. The server should deal additional cards (as per the rules of the game) to each player as requested.

17.21 Develop a Poker card game in which the server application deals cards to each of the client applets. The server should deal additional cards (as per the rules of the game) to each player as requested.

17.22 (Modifications to the Multithreaded Tic-Tac-Toe Program) The programs of Fig. 17.8 and Fig. 17.9 implemented a multithreaded, client/server version of the game Tic-Tac-Toe. Our goal in developing this game was to demonstrate a multithreaded server that could process multiple connections from clients at the same time. The server in the example is really a mediator between the two client applets—it makes sure that each move is valid and that each client moves in the proper order. The server does not determine who won or lost or if there was a draw. Also, there is no capability to allow a new game to be played or to terminate an existing game.

The following is a list of suggested modifications to the multithreaded Tic-Tac-Toe application and applet.

a) Modify the TicTacToeServer class to test for a win, loss or draw on each move in the game. Send a message to each client applet that indicates the result of the game when the game is over.

b) Modify the TicTacToeClient class to display a button that when clicked allows the client to play another game. The button should be enabled only when a game completes. Note that both class TicTacToeClient and class TicTacToeServer must be modified to reset the board and all state information. Also, the other TicTacToeClient should be notified that a new game is about to begin so its board and state can be reset.

c) Modify the TicTacToeClient class to provide a button that allows a client to terminate the program at any time. When the user clicks the button, the server and the other client should be notified. The server should then wait for a connection from another client so a new game can begin.

d) Modify the TicTacToeClient class and the TicTacToeServer class so the winner of a game can choose game piece X or O for the next game. Remember: X always goes first.
e) If you would like to be ambitious, allow a client to play against the server while the server waits for a connection from another client.

17.23  *(3-D Multithreaded Tic-Tac-Toe)* Modify the multithreaded, client/server Tic-Tac-Toe program to implement a three-dimensional 4-by-4-by-4 version of the game. Implement the server application to mediate between the two clients. Display the three-dimensional board as four boards containing four rows and four columns each. If you would like to be ambitious, try the following modifications:

a) Draw the board in a three-dimensional manner.

b) Allow the server to test for a win, loss or draw. Beware! There are many possible ways to win on a 4-by-4-by-4 board!

17.24  *(Networked Morse Code)* Modify your solution to Exercise 10.27 to enable two applets to send Morse Code messages to each other through a multithreaded server application. Each applet should allow the user to type normal characters in JTextArea, translate the characters into Morse Code and send the coded message through the server to the other client. When messages are received, they should be decoded and displayed as normal characters and as Morse Code. The applet should have two JTextAreas: one for displaying the other client’s messages and one for typing.
Multimedia: Images, Animation, Audio and Video

Objectives

- To understand how to get and display images.
- To create animations from sequences of images.
- To customize an animation applet with applet parameters specified in the applet’s HTML document.
- To create image maps.
- To be able to get, play, loop and stop sounds using an AudioClip.

The wheel that squeaks the loudest … gets the grease.
John Billings (Henry Wheeler Shaw)

We’ll use a signal I have tried and found far-reaching and easy to yell. Waa-hoo!
Zane Grey

There is a natural hootchy-kootchy motion to a goldfish.
Walt Disney

Between the motion and the act falls the shadow.
Thomas Stearns Eliot, The Hollow Men
18.1 Introduction

Welcome to what may be the largest revolution in the history of the computer industry. Those of us who entered the field decades ago were interested in using computers primarily to perform arithmetic calculations at high speed. As the computer field evolves, we are beginning to realize that the data-manipulation capabilities of computers are now equally important. The “sizzle” of Java is multimedia, the use of sound, images, graphics and video to make applications “come alive.” Today, many people consider two-dimensional color video to be the “ultimate” in multimedia. Within the decade, we expect all kinds of exciting new three-dimensional applications. Java programmers already can use the Java3D API to create substantial 3D graphics applications. We discuss the Java3D API in our book Advanced Java 2 Platform How to Program.

Multimedia programming offers many new challenges. The field is already enormous and will grow rapidly. People are rushing to equip their computers for multimedia. Most new computers sold today are “multimedia ready,” with CD or DVD drives, audio boards and sometimes with special video capabilities.

Among users who want graphics, two-dimensional graphics no longer suffice. Many people now want three-dimensional, high-resolution, color graphics. True three-dimensional imaging may become available within the next decade. Imagine having ultra-high-resolution, “theater-in-the-round,” three-dimensional television. Sporting and entertainment events will seem to take place on your living room floor! Medical students worldwide will see operations being performed thousands of miles away, as if they were occurring in the same room. People will be able to learn how to drive with extremely realistic driving simulators in their homes before they get behind the wheel. The possibilities are exciting and endless.

Multimedia demands extraordinary computing power. Until recently, affordable computers with this kind of power were not available. Today’s ultrafast processors, like the SPARC Ultra from Sun Microsystems, the Pentium and Itanium from Intel, the Alpha from Compaq Computer Corporation and the processors from MIPS/Silicon Graphics (among others) make effective multimedia possible. The computer and communications industries will be primary beneficiaries of the multimedia revolution. Users will be willing to pay for the faster processors, larger memories and wider communications bandwidths that support demanding multimedia applications. Ironically, users may not have to pay more as fierce competition in these industries drives prices down.
We need programming languages that make creating multimedia applications easy. Most programming languages do not have built-in multimedia capabilities. However, Java provides extensive multimedia facilities that enable you to start developing powerful multimedia applications immediately.

This chapter presents a series of “live-code” examples that cover several interesting multimedia features you will need to build useful applications, including:

1. the basics of manipulating images
2. creating smooth animations
3. customizing an animation applet via parameters supplied from the HTML file that invokes an applet
4. playing audio files with the AudioClip interface
5. creating image maps that can sense when the cursor is over them even without a mouse click

We will continue our coverage of Java’s multimedia capabilities in Chapter 22, where we discuss the Java Media Framework (JMF) and the Java Sound APIs. JMF and Java Sound enable Java programs to play and record audio and video. The JMF even enables Java programs to send audio and video streams—so-called streaming media—across a network or the Internet. The chapter exercises for this chapter and Chapter 22 suggest dozens of challenging and interesting projects and even mention some “million-dollar” ideas that could help you make your fortune! When we were creating these exercises, it seemed that the ideas just kept flowing. Multimedia seems to leverage creativity in ways that we have not experienced with “conventional” computer capabilities.

18.2 Loading, Displaying and Scaling Images

Java’s multimedia capabilities include graphics, images, animations, sounds and video. We begin our multimedia discussion with images.

The applet of Fig. 18.1 demonstrates loading an Image (package java.awt) and loading an ImageIcon (package javax.swing). The applet displays the Image in its original size and scaled to a larger size, using two versions of Graphics method drawImage. The applet also draws the ImageIcon using its method paintIcon. Class ImageIcon is easier than Image to use, because its constructor can receive arguments of several different formats, including a byte array containing the bytes of an image, an Image already loaded in memory, a String representing the location of an image and a URL representing the location of an image.

```java
// Fig. 18.1: LoadImageAndScale.java
// Load an image and display it in its original size
// and scale it to twice its original width and height.
// Load and display the same image as an ImageIcon.
// Java core packages
import java.applet.Applet;
import java.awt.*;
```

Fig. 18.1 Loading and displaying an image in an applet (part 1 of 2).
// Java extension packages
import javax.swing.*;

public class LoadImageAndScale extends JApplet {
    private Image logo1;
    private ImageIcon logo2;

    // load image when applet is loaded
    public void init()
    {
        logo1 = getImage( getDocumentBase(), "logo.gif" );
        logo2 = new ImageIcon( "logo.gif" );
    }

    // display image
    public void paint( Graphics g )
    {
        // draw original image
        g.drawImage( logo1, 0, 0, this );

        // draw image scaled to fit width of applet
        // and height of applet minus 120 pixels
        g.drawImage( logo1, 0, 120,
                     getWidth(), getHeight() - 120, this );

        // draw icon using its paintIcon method
        logo2.paintIcon( this, g, 180, 0 );
    }
}

Fig. 18.1 Loading and displaying an image in an applet (part 2 of 2).
Lines 14 and 15 declare an `Image` reference and an `ImageIcon` reference, respectively. Class `Image` is an `abstract` class; therefore, the applet cannot create an object of class `Image` directly. Rather, the applet must call a method that causes the applet container to load and return the `Image` for use in the program. Class `Applet` (the superclass of `JApplet`) provides a method that does just that. Line 20 in method `init` uses `Applet` method `getImage` to load an `Image` into the applet. This version of `getImage` takes two arguments—the location of the image file and the file name of the image. In the first argument, `Applet` method `getDocumentBase` returns a `URL` representing the location of the image on the Internet (or on your computer if the applet was loaded from your computer). The program assumes that the image is stored in the same directory as the HTML file that invoked the applet. Method `getDocumentBase` returns the location of the HTML file on the Internet as an object of class `URL`. The second argument specifies an image file name. Java supports several image formats, including `Graphics Interchange Format (GIF)`, `Joint Photographic Experts Group (JPEG)` and `Portable Network Graphics (PNG)`. File names for each of these types end with `.gif`, `.jpg` (or `.jpeg`) and `.png`, respectively.

**Portability Tip 18.1**

Class `Image` is an `abstract` class and, as a result, programs cannot instantiate `Image` objects. To achieve platform independence, the Java implementation on each platform provides its own subclass of `Image` to store image information.

When line 20 invokes method `getImage` to set up loading of the image from the local computer (or downloading of the image from the Internet). When the image is required by the program, the image is loaded in a separate thread of execution. This enables the program to continue execution while the image loads. [Note: If the requested file is not available, method `getImage` does not indicate an error.]

Class `ImageIcon` is not an `abstract` class; therefore, a program can create an `ImageIcon` object. Line 21 in method `init` creates an `ImageIcon` object that loads the same `logo.gif` image. Class `ImageIcon` provides several constructors that enable programs to initialize `ImageIcon` objects with images from the local computer or with images stored on the Internet.

The applet’s `paint` method (lines 25–37) displays the images. Line 28 uses `Graphics` method `drawImage` to display an `Image`. Method `drawImage` receives four arguments. The first argument is a reference to the `Image` object to display (`logo1`). The second and third arguments are the `x`- and `y`-coordinates at which to display the image on the applet; the coordinates indicate the upper-left corner of the image. The last argument is a reference to an `ImageObserver` object. Normally, the `ImageObserver` is the object on which the program displays the image. An `ImageObserver` can be any object that implements interface `ImageObserver`. Class `Component` (one of class `Applet`’s indirect superclasses) implements interface `ImageObserver`. Therefore, all `Components` (including our applet) are `ImageObservers`. The `ImageObserver` argument is important when displaying large images that require a long time to download from the Internet. It is possible that a program will execute the code that displays the image before the image downloads completely. The `ImageObserver` is notified to update the displayed image as the remainder of the image loads. When executing this applet, watch carefully as pieces of the image display while the image loads. [Note: On faster computers, you might not notice this effect.]

Lines 32–33 use another version of `Graphics` method `drawImage` to output a `scaled` version of the image. The fourth and fifth arguments specify the `width` and `height` of
the image for display purposes. Method `drawImage` scales the image to fit the specified width and height. In this example, the fourth argument indicates that the width of the scaled image should be the width of the applet, and the fifth argument indicates that the height should be 120 pixels less than the height of the applet. Line 33 determines the width and height of the applet by calling methods `getWidth` and `getHeight` (inherited from class `Component`).

Line 36 uses `ImageIcon` method `paintIcon` to display the image. The method requires four arguments—a reference to the `Component` on which to display the image, a reference to the `Graphics` object that will render the image, the x-coordinate of the upper-left corner of the image and the y-coordinate of the upper-left corner of the image.

If you compare the two techniques for loading and displaying images in this example, you can see that using `ImageIcon` is simpler. You can create objects of class `ImageIcon` directly, and there is no need to use an `ImageObserver` reference when displaying the image. For this reason, we use class `ImageIcon` for the remainder of the chapter. [Note: Class `ImageIcon`'s `paintIcon` method does not allow scaling of an image. However, the class provides method `getImage`, which returns an `Image` reference that `Graphics` method `drawImage` can use to display a scaled image.]

### 18.3 Animating a Series of Images

The next example demonstrates animating a series of images that are stored in an array. The application uses the same techniques to load and display `ImageIcon`s as shown in Fig. 18.1. The animation presented in Fig. 18.2 is designed as a subclass of `JPanel` (called `LogoAnimator`) that can be attached to an application window or possibly to a JApplet. Class `LogoAnimator` also defines a `main` method (defined at lines 96–117) to execute the animation as an application. Method `main` defines an instance of class `JFrame` and attaches a `LogoAnimator` object to the `JFrame` to display the animation.

```
// Fig. 18.2: LogoAnimator.java
// Animation a series of images

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

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

public class LogoAnimator extends JPanel
    implements ActionListener {

    protected ImageIcon images[];       // array of images
    protected int totalImages = 30,     // number of images
        currentImage = 0,     // current image index
        animationDelay = 50,  // millisecond delay
        width,                // image width
        height;               // image height
```
protected String imageName = "deitel";  // base image name
protected Timer animationTimer;  // Timer drives animation

// initialize LogoAnimator by loading images
public LogoAnimator()
{
    initializeAnimation();
}

// initialize animation
protected void initializeAnimation()
{
    images = new ImageIcon[ totalImages ];
    // load images
    for ( int count = 0; count < images.length; ++count )
        images[ count ] = new ImageIcon( getClass().getResource("images/" + imageName + count + ".gif" ) );
    width = images[ 0 ].getIconWidth();   // get icon width
    height = images[ 0 ].getIconHeight(); // get icon height
}

// display current image
public void paintComponent( Graphics g )
{
    super.paintComponent( g );
    images[ currentImage ].paintIcon( this, g, 0, 0 );
    currentImage = ( currentImage + 1 ) % totalImages;
}

// respond to Timer's event
public void actionPerformed( ActionEvent actionEvent )
{
    repaint();  // repaint animator
}

// start or restart animation
public void startAnimation()
{
    if ( animationTimer == null ) {
        currentImage = 0;
        animationTimer = new Timer( animationDelay, this );
        animationTimer.start();
    }
    else // continue from last image displayed
    if ( ! animationTimer.isRunning() )
        animationTimer.restart();
}

Fig. 18.2 Animating a series of images (part 2 of 3).
// stop animation timer
public void stopAnimation()
{
    animationTimer.stop();
}

// return minimum size of animation
public Dimension getMinimumSize()
{
    return getPreferredSize();
}

// return preferred size of animation
public Dimension getPreferredSize()
{
    return new Dimension( width, height );
}

// execute animation in a JFrame
public static void main( String args[] )
{

    // create LogoAnimator
    LogoAnimator animation = new LogoAnimator();

    // set up window
    JFrame window = new JFrame( "Animator test" );
    Container container = window.getContentPane();
    container.add( animation );
    window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

    // size and display window
    window.pack();
    Insets insets = window.getInsets();
    window.setSize( animation.getPreferredSize().width +
        insets.left + insets.right,
        animation.getPreferredSize().height +
        insets.top + insets.bottom );
    window.setVisible( true );
    animation.startAnimation();  // begin animation

}  // end method main

} // end class LogoAnimator

Fig. 18.2 Animating a series of images (part 3 of 3).
Class **LogoAnimator** maintains an array of **ImageIcons** that are loaded in method **initializeAnimation** (lines 32–43), which is called from the constructor. As the **for** structure (lines 37–39) creates each **ImageIcon** object, the **ImageIcon** constructor loads one of the animation’s 30 images. The constructor argument uses **String** concatenation to assemble the file name from the pieces "images/", **imageName**, **count** and ".gif". Each of the images in the animation is in a file called **deitel#{gif**, where # is a value in the range 0–29 specified by the loop’s control variable **count**. Lines 41–42 determine the width and height of the animation from the size of the first image in array **images**.

**Performance Tip 18.1**

It is more efficient to load the frames of the animation as one image than to load each image separately. (A painting program can be used to combine the frames of the animation into one image. If the images are being loaded from the Web, every image loaded requires a separate connection to the site containing the images.

**Performance Tip 18.2**

Loading all the frames of an animation as one large image can force your program to wait to begin displaying the animation.

After the **LogoAnimator** constructor loads the images, method **main** sets up the window in which the animation will appear and calls **startAnimation** (defined at lines 61–71) to begin the animation. The animation is driven by an instance of class **Timer** (package **javax.swing**). A **Timer** generates **ActionEvents** at a fixed interval in milliseconds (normally specified as an argument to the **Timer**’s constructor) and notifies all of its **ActionListeners** that the event occurred. Lines 63–67 determine whether the **Timer** reference **animationTimer** is **null**. If so, line 64 sets **currentImage** to 0, to indicate that the animation should begin with the image in the first element of array **images**. Line 65 assigns a new **Timer** object to **animationTimer**. The **Timer** constructor receives two arguments—the delay in milliseconds (**animationDelay** is 50 in this example) and the **ActionListener** that will respond to the **Timer**’s **ActionEvents**. Class **LogoAnimator** implements **ActionListener** so line 65 specifies **this** as the listener. Line 66 starts the **Timer** object. Once started, **animationTimer** will generate an **ActionEvent** every 50 milliseconds. Lines 69–70 enable a program to restart an animation that the program stopped previously. For example, to make an animation “browser friendly” in an applet, the animation should stop when the user switches Web pages. If the user returns to the Web page with the animation, method **startAnimation** can be called to restart the animation. The **if** condition at line 73 uses **Timer** method **isRunning** to determine whether the **Timer** is running (i.e., generating events). If it is not running, line 70 calls **Timer** method **restart** to indicate that the **Timer** should start generating events again.

In response to every **Timer** event in this example, the program calls method **actionPerformed** (lines 55–58). Line 57 calls **LogoAnimator**’s **repaint** method to schedule a call to the **LogoAnimator**’s **update** method (inherited from class **JPanel**) which, in turn, calls **LogoAnimator**’s **paintComponent** method (lines 46–52). Remember that any subclass of **JComponent** that performs drawing should do so in its **paintComponent** method. As mentioned in Chapter 13, the first statement in any **paintComponent** method should be a call to the superclass’s **paintComponent** method to ensure that Swing components are displayed correctly.
Lines 50–51 paint the `ImageIcon` at element `currentImage` in the array and prepare for the next image to be displayed by incrementing `currentImage` by 1. Notice the modulus calculation to ensure that the value of `currentImage` is set to 0 when it is incremented past 29 (the last element subscript in the array).

Method `stopAnimation` (lines 74–77) stops the animation with line 76, which calls `Timer` method `stop` to indicate that the `Timer` should stop generating events. This prevents `actionPerformed` from calling `repaint` to initiate the painting of the next image in the array.

**Software Engineering Observation 18.1**

When creating an animation for use in an applet, provide a mechanism for disabling the animation when the user browses a new Web page separate from the page on which the animation applet resides.

Methods `getMinimumSize` (lines 80–83) and `getPreferredSize` (lines 86–89) override the corresponding methods inherited from class `Component` and enable layout managers to determine the appropriate size of a `LogoAnimator` in a layout. In this example, the images are 160 pixels wide and 80 pixels tall, so method `getPreferredSize` returns a `Dimension` object containing the numbers 160 and 80. Method `getMinimumSize` simply calls `getPreferredSize` (a common programming practice).

Lines 107–112 of `main` size the application window based on the `LogoAnimator`’s preferred size and the window’s `insets`. The insets specify the number of pixels for the window’s top, bottom, left and right borders. Using the insets of the window ensures that the window’s `client area` is large enough to display the `LogoAnimator` correctly. The client area is the part of a window in which the window displays GUI components. The program obtains the window’s insets by calling method `getInsets` at line 107. The method returns an `Insets` object that contains public data members `top`, `bottom`, `left` and `right`.

### 18.4 Customizing LogoAnimator via Applet Parameters

When browsing the World Wide Web, you often will come across applets that are in the public domain—you can use them free of charge on your own Web pages (normally in exchange for crediting the applet’s creator). One common feature of such applets is the ability to be customized via parameters that are supplied from the HTML file that invokes the applet. For example, the HTML

```html
<html>
<applet code = "LogoApplet.class" width = 400 height = 400>
<param name = "totalimages" value = "30">
<param name = "imagename" value = "deitel">
<param name = "animationdelay" value = "200">
</applet>
</html>
```

from file `LogoApplet.html`, invokes the applet `LogoApplet` (Fig. 18.4) and specifies three parameters. The `param` tag lines must appear between the starting and ending `applet` tags. Each parameter has a `name` and a `value`. `Applet` method `getParameter` returns a `String` representing the `value` associated with a specific parameter `name`. The argument to `getParameter` is a `String` containing the name of the parameter in the `param` tag. For example, the statement
parameter = getParameter( "animationdelay" );

gets the value 200 associated with parameter animationdelay and assigns it to String reference parameter. If there is not a param tag containing the specified parameter name, getParameter returns null.

Class LogoAnimator2 (Fig. 18.3) extends class LogoAnimator and adds a constructor that takes three arguments—the total number of images in the animation, the delay between displaying images and the base image name. Class LogoApplet (Fig. 18.4) allows Web page designers to customize the animation to use their own images. Three parameters are provided in the applet’s HTML document. Parameter animationdelay is the number of milliseconds to sleep between displaying images. This value will be converted to an integer and used as the value for instance variable sleepTime. Parameter imagename is the base name of the images to be loaded. This String will be assigned to instance variable imageName. The applet assumes that the images are in a subdirectory named images that can be found in the same directory as the applet. The applet also assumes that the image file names are numbered from 0 (as in Fig. 18.2). Parameter totalimages represents the total number of images in the animation. Its value will be converted to an integer and assigned to instance variable totalImages.

```
1 // Fig. 18.3: LogoAnimator2.java
2 // Animating a series of images
3
4 // Java core packages
5 import java.awt.*;
6
7 // Java extension packages
8 import javax.swing.*;
9
10 public class LogoAnimator2 extends LogoAnimator {
11
12 // default constructor
13 public LogoAnimator2()
14 {
15    super();
16 }
17
18 // new constructor to support customization
19 public LogoAnimator2( int count, int delay, String name )
20 {
21    totalImages = count;
22    animationDelay = delay;
23    imageName = name;
24    initializeAnimation();
25 }
26
27
```

Fig. 18.3 LogoAnimator2 subclass of LogoAnimator (Fig. 18.2) adds a constructor for customizing the number of images, animation delay and base image name (part 1 of 2).
```java
// start animation as application in its own window
public static void main( String args[] )
{
    // create LogoAnimator
    LogoAnimator2 animation = new LogoAnimator2();

    // set up window
    JFrame window = new JFrame( "Animator test" );
    Container container = window.getContentPane();
    container.add( animation );
    window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

    // size and display window
    window.pack();
    Insets insets = window.getInsets();
    window.setSize( animation.getPreferredSize().width +
                    insets.left + insets.right,
                    animation.getPreferredSize().height +
                    insets.top + insets.bottom );
    window.setVisible( true );
    animation.startAnimation();  // begin animation
}
}  // end method main
} // end class LogoAnimator2
```

Fig. 18.3 LogoAnimator2 subclass of LogoAnimator (Fig. 18.2) adds a constructor for customizing the number of images, animation delay and base image name (part 2 of 2).

```java
// Fig. 18.4: LogoApplet.java
// Customizing an applet via HTML parameters.
//
// HTML parameter "animationdelay" is an int indicating milliseconds to sleep between images (default 50).
//
// HTML parameter "imagename" is the base name of the images that will be displayed (i.e., "deitel" is the base name for images "deitel0.gif," "deitel1.gif," etc.). The applet assumes that images are in an "images" subdirectory of the directory in which the applet resides.
//
// HTML parameter "totalimages" is an integer representing the total number of images in the animation. The applet assumes images are numbered from 0 to totalimages - 1 (default 30).
//
// Java core packages
import java.awt.*;
```

Fig. 18.4 Customizing an animation applet via the param HTML tag (part 1 of 2).
// Java extension packages
import javax.swing.*;

public class LogoApplet extends JApplet {

    // obtain parameters from HTML and customize applet
    public void init() {
        String parameter;

        // get animation delay from HTML document
        parameter = getParameter( "animationdelay" );

        int animationDelay = ( parameter == null ?
            50 : Integer.parseInt( parameter ) );

        // get base image name from HTML document
        String imageName = getParameter( "imagename" );

        // get total number of images from HTML document
        parameter = getParameter( "totalimages" );

        int totalImages = ( parameter == null ?
            0 : Integer.parseInt( parameter ) );

        // create instance of LogoAnimator
        LogoAnimator2 animator;

        if ( imageName == null || totalImages == 0 )
            animator = new LogoAnimator2();
        else
            animator = new LogoAnimator2( totalImages, animationDelay, imageName );

        // attach animator to applet and start animation
        getContentPane().add( animator );
        animator.startAnimation();

    }  // end method init

} // end class LogoApplet

Fig. 18.4 Customizing an animation applet via the param HTML tag (part 2 of 2).
Class **LogoApplet** (Fig. 18.4) defines an **init** method in which the three HTML parameters are read with **Applet** method **getParameter** (lines 31, 37 and 40). After the applet reads these parameters and the two integer parameters are converted to **int** values, the **if/else** structure at lines 48 through 51 creates a **LogoAnimator2** and calls its three-argument constructor. If the **imageName** is **null** or **totalImages** is 0, the applet calls the default **LogoAnimator2** constructor and uses the default animation. Otherwise, the applet passes **totalImages, animationDelay** and **imageName** to the three-argument **LogoAnimator2** constructor, and the constructor uses those arguments to customize the animation. **LogoAnimator2**’s three-argument constructor invokes **LogoAnimator**’s **initializeAnimation** method to load the images and determine the width and height of the animation.

### 18.5 Image Maps

Image maps are a common technique used to create interactive Web pages. An image map is an image that has **hot areas** that the user can click to accomplish a task, such as loading a different Web page into a browser. When the user positions the mouse pointer over a hot area, normally a descriptive message appears in the status area of the browser or in a pop-up window.

Figure 18.5 loads an image containing several of the common tip icons used throughout this book. The program allows the user to position the mouse pointer over an icon and display a descriptive message for the icon. Event handler **mouseMoved** (lines 42–46) takes the mouse coordinates and passes them to method **translateLocation** (lines 63–76). Method **translateLocation** tests the coordinates to determine the icon over which the mouse was positioned when the **mouseMoved** event occurred. Method **translateLocation** then returns a message indicating what the icon represents. This message is displayed in the **appletviewer**’s (or browser’s) status bar.

```java
// Fig. 18.5: ImageMap.java
// Demonstrating an image map.

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

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

public class ImageMap extends JApplet {
    private ImageIcon mapImage;
    private String captions[] = {
        "Common Programming Error",
        "Good Programming Practice",
        "Graphical User Interface Tip", "Performance Tip",
        "Portability Tip", "Software Engineering Observation",
        "Testing and Debugging Tip" 
    };
}
```

**Fig. 18.5** Demonstrating an image map (part 1 of 4).
// set up mouse listeners
public void init()
{
    addMouseListener(
        new MouseAdapter() {
            // indicate when mouse pointer exits applet area
            public void mouseExited( MouseEvent event )
            {
                showStatus( "Pointer outside applet" );
            }
        }  // end anonymous inner class
    ); // end addMouseListener method call

    addMouseMotionListener(
        new MouseMotionAdapter() {
            // determine icon over which mouse appears
            public void mouseMoved( MouseEvent event )
            {
                showStatus( translateLocation( event.getX(), event.getY() ) );
            }
        }  // end anonymous inner class
    ); // end addMouseMotionListener method call

    mapImage = new ImageIcon( "icons.png" );
}  // end method init

// display mapImage
public void paint( Graphics g )
{
    mapImage.paintIcon( this, g, 0, 0 );
}

// return tip caption based on mouse coordinates
public String translateLocation( int x, int y )
{
    // if coordinates outside image, return immediately
    if ( x >= mapImage.getIconWidth() ||
        y >= mapImage.getIconHeight() )
        return "";

    // determine icon number (0 - 6)
    int iconWidth = mapImage.getIconWidth() / 7;
    int iconNumber = x / iconWidth;

Fig. 18.5 Demonstrating an image map (part 2 of 4).
// return appropriate icon caption
return captions[iconNumber];

} // end class ImageMap

Fig. 18.5 Demonstrating an image map (part 3 of 4).
Clicking in this applet will not cause any action. In Chapter 17, Networking, we discussed the techniques required to load another Web page into a browser via URLs and the AppletContext interface. Using those techniques, this applet could associate each icon with a URL that the browser would display when the user clicks the icon.

18.6 Loading and Playing Audio Clips

Java programs can manipulate and play audio clips. It is easy for users to capture their own audio clips, and there are many clips available in software products and over the Internet. Your system needs to be equipped with audio hardware (speakers and a sound board) to be able to play the audio clips.

Java provides several mechanisms for playing sounds in an applet. The two simplest methods are the Applet's play method and the play method from the AudioClip interface. Additional audio capabilities are discussed in Chapter 22. If you would like to play a sound once in a program, the Applet method play loads the sound and plays it once; the sound is marked for garbage collection after it plays. The Applet method play has two forms:

```
public void play( URL location, String soundFileName );
public void play( URL soundURL );
```

The first version loads the audio clip stored in file soundFileName from location and plays the sound. The first argument is normally a call to the applet’s getDocument-
Base or `getCodeBase` method. Method `getDocumentBase` indicates the location of the HTML file that loaded the applet (if the applet is in a package, this indicates the location of the package or JAR file containing the package). Method `getCodeBase` indicates the location of the applet’s `.class` file. The second version of method `play` takes a URL that contains the location and the file name of the audio clip. The statement

```java
play( getDocumentBase(), "hi.au" );
```

loads the audio clip in file `hi.au` and plays it once.

The sound engine that plays the audio clips supports several audio file formats, including Sun Audio file format (.au extension), Windows Wave file format (.wav extension), Macintosh AIFF file format (.aif or .aiff extension) and Musical Instrument Digital Interface (MIDI) file format (.mid or .rmi extensions). The Java Media Framework (JMF) and Java Sound APIs support additional formats.

The program of Fig. 18.6 demonstrates loading and playing an `AudioClip` (package `java.applet`). This technique is more flexible than `Applet` method `play`. An applet can use an AudioClip to store audio for repeated use throughout the program’s execution. `Applet` method `getAudioClip` has two forms that take the same arguments as method `play` described previously. Method `getAudioClip` returns a reference to an `AudioClip`. An `AudioClip` has three methods—`play`, `loop` and `stop`. Method `play` plays the audio once. Method `loop` continuously loops the audio clip in the background. Method `stop` terminates an audio clip that is currently playing. In the program, each of these methods is associated with a button on the applet.

Lines 62–63 in the applet’s `init` method use `getAudioClip` to load two audio files—a Windows Wave file (`welcome.wav`) and a Sun Audio file (`hi.au`). The user can select which audio clip to play from `JComboBox chooseSound`. Notice that the applet’s `stop` method is overridden at lines 69–72. When the user switches Web pages, the applet container calls the applet’s `stop` method. This enables the applet to stop playing the audio clip. Otherwise, the audio clip continues to play in the background—even if the applet is not displayed in the browser. This is not really a problem, but it can be annoying to the user if the audio clip is looping. The `stop` method is provided here as a convenience to the user.

**Good Programming Practice 18.1**

When playing audio clips in an applet or application, provide a mechanism for the user to disable the audio.

```java
// Fig. 18.6: LoadAudioAndPlay.java
// Load an audio clip and play it.

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

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

// Fig. 18.6 Loading and playing an AudioClip (part 1 of 3).
```
public class LoadAudioAndPlay extends JApplet {
    private AudioClip sound1, sound2, currentSound;
    private JButton playSound, loopSound, stopSound;
    private JComboBox chooseSound;

    // load the image when the applet begins executing
    public void init() {
        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

        String choices[] = { "Welcome", "Hi" };
        chooseSound = new JComboBox( choices );

        chooseSound.addItemListener(
            new ItemListener() {
                public void itemStateChanged( ItemEvent e ) {
                    currentSound.stop();
                    currentSound = chooseSound.getSelectedIndex() == 0 ? sound1 : sound2;
                }
            }); // end addItemListener method call
        container.add( chooseSound );

        // set up button event handler and buttons
        ButtonHandler handler = new ButtonHandler();

        playSound = new JButton( "Play" );
        playSound.addActionListener( handler );
        container.add( playSound );

        loopSound = new JButton( "Loop" );
        loopSound.addActionListener( handler );
        container.add( loopSound );

        stopSound = new JButton( "Stop" );
        stopSound.addActionListener( handler );
        container.add( stopSound );

        // load sounds and set currentSound
        sound1 = getAudioClip( getDocumentBase(), "welcome.wav" );
        sound2 = getAudioClip( getDocumentBase(), "hi.au" );
    }
} // end anonymous inner class

Fig. 18.6 Loading and playing an AudioClip (part 2 of 3).
18.7 Internet and World Wide Web Resources

This section presents several Internet and Web resources for multimedia-related sites. (Additional resources are provided in Chapter 22.)

www.nasa.gov/gallery/index.html
The NASA multimedia gallery contains a wide variety of images, audio clips and video clips that you can download and use to test your Java multimedia programs.

sunsite.sut.ac.jp/multimed/
The Sunsite Japan Multimedia Collection also provides a wide variety of images, audio clips and video clips that you can download for educational purposes.

The Australian National Botanic Gardens Web site provides links to sounds of many animals. Try, for example, the Common Birds link.
www.thefreesite.com
*TheFreeSite.com* has links to free sounds and clip art.

www.soundcentral.com
*SoundCentral* provides audio clips in WAV, AU, AIFF and MIDI formats.

www.animationfactory.com
The *Animation Factory* provides thousands of free GIF animations for personal use.

www.clipart.com
*ClipArt.com* contains links to Web sites that provide free art.

www.pngart.com
PNGART.com provides over 50,000 free images in PNG format, in an effort to help this newer image format gain popularity.

developer.java.sun.com/developer/techDocs/hi/repository
The *Java Look-and-Feel Graphics Repository* provides standard images for use in a Swing GUI.

**SUMMARY**

- **Applet** method `getImage` loads an *Image*. One version of `getImage` takes two arguments—a location where the image is stored, and the file name of the image.

- **Applet** method `getDocumentBase` returns the location of the applet’s HTML file on the Internet as an object of class *URL* (package *java.net*).

- Java supports several image formats, including Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) and Portable Network Graphics (PNG). File names for each of these types end with `.gif`, `.jpg` (or `.jpeg`) or `.png`, respectively.

- Class *ImageIcon* provides constructors that allow an *ImageIcon* object to be initialized with an image from the local computer or with an image stored on a Web server on the Internet.

- **Graphics** method `drawImage` receives four arguments—a reference to the *Image* object in which the image is stored, the x- and y-coordinates where the image should be displayed and a reference to an *ImageObserver* object.

- Another version of **Graphics** method `drawImage` outputs a *scaled* image. The fourth and fifth arguments specify the width and height of the image for display purposes.

- Interface *ImageObserver* is implemented by class *Component* (an indirect superclass of *Applet*). *ImageObservers* are notified to update an image that was displayed as the remainder of the image is loaded.

- **ImageIcon** method `paintIcon` displays the *ImageIcon*’s image. The method requires four arguments—a reference to the *Component* on which the image will be displayed, a reference to the **Graphics** object used to render the image, the x-coordinate of the upper-left corner of the image and the y-coordinate of the upper-left corner of the image.

- Class *ImageIcon*’s `paintIcon` method does not allow scaling of an image. The class provides method `getImage`, which returns an *Image* reference that can be used with **Graphics** method `drawImage` to display a scaled version of an image.

- **Timer** objects generate **ActionEvents** at fixed intervals in milliseconds and notify their registered **ActionListeners** that the events occurred. The **Timer** constructor receives two arguments—the delay in milliseconds and the **ActionListener**. **Timer** method `start` indicates that the **Timer** should start generating events. **Timer** method `stop` indicates that the **Timer** should stop generating events. **Timer** method `restart` indicates that the **Timer** should start generating events again.
• Applets can be customized via parameters (the `<param>` tag) that are supplied from the HTML file that invokes the applet. The `<param>` tag lines must appear between the starting `applet` tag and the ending `applet` tag. Each parameter has a `name` and a `value`.

• **Applet** method `getParameter` gets the `value` associated with a specific parameter and returns the `value` as a `String`. The argument passed to `getParameter` is a `String` containing the name of the parameter in the `param` tag. If there is no `param` tag containing the specified parameter, `getParameter` returns `null`.

• An image map is an image that has hot areas that the user can click to accomplish a task, such as loading a different Web page into a browser.

• **Applet** method `play` has two forms:

  ```java
  public void play( URL location, String soundFileName );
  public void play( URL soundURL );
  ```

  One version loads the audio clip stored in file `soundFileName` from `location` and plays the sound; the other takes a `URL` that contains the location and the file name of the audio clip.

• **Applet** method `getDocumentBase` indicates the location of the HTML file that loaded the applet. Method `getCodeBase` indicates where the `.class` file for an applet is located.

• The sound engine that plays audio clips supports several audio file formats, including Sun Audio file format (`.au` extension), Windows Wave file format (`.wav` extension), Macintosh AIFF file format (`.aif` or `.aiff` extension) and Musical Instrument Digital Interface (MIDI) file format (`.mid` or `.rmi` extensions). The Java Media Framework (JMF) supports other additional formats.

• **Applet** method `getAudioClip` has two forms that take the same arguments as the `play` method. Method `getAudioClip` returns a reference to an `AudioClip`. `AudioClip`s have three methods—`play`, `loop` and `stop`. Method `play` plays the audio once. Method `loop` continuously loops the audio clip. Method `stop` terminates an audio clip that is currently playing.

**TERMINOLOGY**

| **.aif** file name extension | **getImage** method of **ImageIcon** |
| **.aiff** file name extension | **getParameter** method of **Applet** |
| **.au** file name extension | **getWidth** method of **Component** |
| **.gif** file name extension | **graphics** |
| **.jpeg** file name extension | Graphics Interchange Format (GIF) |
| **.jpg** file name extension | height of an image |
| **.mid** file name extension | hot area of an image map |
| **.rmi** file name extension | **Image** class |
| **.wav** file name extension | image map |
| animating a series of images | **ImageIcon** class |
| animation | **ImageObserver** interface |
| audio clip | images |
| customize an applet | information button |
| **drawImage** method of **Graphics** | Joint Photographic Experts Group (JPEG) |
| **getAudioClip** method of **Applet** | **loop** method of interface **AudioClip** |
| **getCodeBase** method of **Applet** | Macintosh AIFF file (`.aif` or `.aiff`) |
| **getDocumentBase** method of **Applet** | multimedia |
| **getHeight** method of **Component** | Musical Instrument Digital Interface (MIDI) |
| **getIconHeight** method of **ImageIcon** | mute button |
| **getIconWidth** method of **ImageIcon** | name attribute of **param** tag |
| **getImage** method of **Applet** | **paintIcon** method of class **ImageIcon** |
SELF-REVIEW EXERCISES

18.1 Fill in the blanks in each of the following statements:
   a) Applet method _________ loads an image into an applet.
   b) Applet method _________ returns as an object of class URL the location on the Internet of the HTML file that invoked the applet.
   c) Graphics method _________ displays an image on an applet.
   d) Java provides two mechanisms for playing sounds in an applet—the Applet’s play method and the play method from the _________ interface.
   e) An _________ is an image that has hot areas that the user can click to accomplish a task such as loading a different Web page.
   f) Method _________ of class ImageIcon displays the ImageIcon’s image.
   g) Java supports several image formats, including _________, _________ and _________.

18.2 State whether each of the following is true or false. If false, explain why.
   a) A sound will be garbage collected as soon as it has finished playing.
   b) Class ImageIcon provides constructors that allow an ImageIcon object to be initialized only with an image from the local computer.
   c) Applet method getParameter gets the value associated with a specific HTML parameter and returns the value as a String.

ANSWERS TO SELF-REVIEW EXERCISES


18.2 a) False. The sound will be marked for garbage collection (if it is not referenced by an AudioClip) and will be garbage collected when the garbage collector is able to run. b) False. ImageIcon can load images from the Internet as well. c) True.

EXERCISES

18.3 Describe how to make an animation “browser friendly.”
18.4 Describe the Java methods for playing and manipulating audio clips.
18.5 How can Java applets be customized with information from an HTML file?
18.6 Explain how image maps are used. List 10 examples in which image maps are used.
18.7 (Randomly Erasing an Image) Suppose an image is displayed in a rectangular screen area. One way to erase the image is simply to set every pixel to the same color immediately, but this is a dull visual effect. Write a Java program that displays an image and then erases it by using random-
number generation to select individual pixels to erase. After most of the image is erased, erase all of the remaining pixels at once. You can refer to individual pixels by having a line that starts and ends at the same point. You might try several variants of this problem. For example, you might display lines randomly or display shapes randomly to erase regions of the screen.

18.8 (Text Flasher) Create a Java program that repeatedly flashes text on the screen. Do this by alternating the text with a plain background-color image. Allow the user to control the “blink speed” and the background color or pattern.

18.9 (Image Flasher) Create a Java program that repeatedly flashes an image on the screen. Do this by alternating the image with a plain background-color image.

18.10 (Digital Clock) Implement a program that displays a digital clock on the screen. You might add options to scale the clock; display day, month and year; issue an alarm; play certain audios at designated times and the like.

18.11 (Calling Attention to an Image) If you want to emphasize an image, you might place a row of simulated light bulbs around your image. You can let the light bulbs flash in unison, or you can let them fire on and off in sequence one after the other.

18.12 (Image Zooming) Create a program that enables you to zoom in on, or away from, an image.

SPECIAL SECTION: CHALLENGING MULTIMEDIA PROJECTS

The preceding exercises are keyed to the text and designed to test the reader’s understanding of fundamental multimedia concepts. This section includes a collection of advanced multimedia projects. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects. [Note: Solutions are not provided for these exercises.]

18.13 (Animation) Create a general purpose Java animation program. Your program should allow the user to specify the sequence of frames to be displayed, the speed at which the images are displayed, audios that should be played while the animation is running and so on.

18.14 (Limericks) Modify the limerick-writing program you wrote in Exercise 10.10 to sing the limericks your program creates.

18.15 (Random Inter-Image Transition) This provides a nice visual effect. If you are displaying one image in a given area on the screen and you would like to transition to another image in the same screen area, store the new screen image in an off-screen buffer and randomly copy pixels from the new image to the display area, overlaying the previous pixels at those locations. When the vast majority of the pixels have been copied, copy the entire new image to the display area to be sure you are displaying the complete new image. To implement this program, you may need to use the Pixel-Grabber and MemoryImageSource classes (see the Java API documentation for descriptions of these classes). You might try several variants of this problem. For example, try selecting all the pixels in a randomly selected straight line or shape in the new image, and overlay those pixels above the corresponding positions of the old image.

18.16 (Background Audio) Add background audio to one of your favorite applications by using the loop method of class AudioClip to play the sound in the background while you interact with your application in the normal way.

18.17 (Scrolling Marquee Sign) Create a Java program that scrolls dotted characters from right to left (or from left to right if that is appropriate for your language) across a Marquee-like display sign. As an option, display the text in a continuous loop, so that after the text disappears at one end it reappears at the other end.
18.18 **(Scrolling Image Marquee)** Create a Java program that scrolls an image across a Marquee screen.

18.19 **(Analog Clock)** Create a Java program that displays an analog clock with hour, minute and second hands that move appropriately as the time changes.

18.20 **(Dynamic Audio and Graphical Kaleidoscope)** Write a kaleidoscope program that displays reflected graphics to simulate the popular children’s toy. Incorporate audio effects that “mirror” your program’s dynamically changing graphics.

18.21 **(Automatic Jigsaw Puzzle Generator)** Create a Java jigsaw puzzle generator and manipulator. Your user specifies an image. Your program loads and displays the image. Your program then breaks the image into randomly selected shapes and shuffles the shapes. The user then uses the mouse to move the puzzle pieces around to solve the puzzle. Add appropriate audio sounds as the pieces are being moved around and snapped back into place. You might keep tabs on each piece and where it really belongs and then use audio effects to help the user get the pieces into the correct positions.

18.22 **(Maze Generator and Walker)** Develop a multimedia-based maze generator and traverser program based on the maze programs you wrote in Exercise 7.38–Exercise 7.40. Let the user customize the maze by specifying the number of rows and columns and by indicating the level of difficulty. Have an animated mouse walk the maze. Use audio to dramatize the movement of your mouse character.

18.23 **(One-Armed Bandit)** Develop a multimedia simulation of a one-armed bandit. Have three spinning wheels. Place various fruits and symbols on each wheel. Use true random-number generation to simulate the spinning of each wheel and the stopping of each wheel on a symbol.

18.24 **(Horse Race)** Create a Java simulation of a horse race. Have multiple contenders. Use audios for a race announcer. Play the appropriate audios to indicate the correct status of each of the contenders throughout the race. Use audios to announce the final results. You might try to simulate the kinds of horse-racing games that are often played at carnivals. The players get turns at the mouse and have to perform some skill-oriented manipulation with the mouse to advance their horses.

18.25 **(Shuffleboard)** Develop a multimedia-based simulation of the game of shuffleboard. Use appropriate audio and visual effects.

18.26 **(Game of Pool)** Create a multimedia-based simulation of the game of pool. Each player takes turns using the mouse to position a pool stick and to hit the stick against the ball at the appropriate angle to try to get the pool balls to fall into the pockets. Your program should keep score.

18.27 **(Artist)** Design a Java art program that will give an artist a great variety of capabilities to draw, use images, use animations, etc., to create a dynamic multimedia art display.

18.28 **(Fireworks Designer)** Create a Java program that someone might use to create a fireworks display. Create a variety of fireworks demonstrations. Then orchestrate the firing of the fireworks for maximum effect.

18.29 **(Floor Planner)** Develop a Java program that will help someone arrange furniture in his or her home. Add features that enable the person to achieve the best possible arrangement.

18.30 **(Crossword)** Crossword puzzles are among the most popular pastimes. Develop a multimedia-based crossword-puzzle program. Your program should enable the player to place and erase words easily. Tie your program to a large computerized dictionary. Your program also should be able to suggest words based on which letters have already been filled in. Provide other features that will make the crossword-puzzle enthusiast’s job easier.

18.31 **(15 Puzzle)** Write a multimedia-based Java program that enables the user to play the game of 15. There is a 4-by-4 board for a total of 16 slots. One of the slots is empty. The other slots are occupied by 15 tiles numbered 1 through 15. Any tile next to the currently empty slot can be moved into
that slot by clicking on the tile. Your program should create the board with the tiles out of order. The goal is to arrange the tiles into sequential order, row by row.

18.32  *(Reaction Time/Reaction Precision Tester)* Create a Java program that moves a randomly created shape around the screen. The user moves the mouse to catch and click on the shape. The shape’s speed and size can be varied. Keep statistics on how much time the user typically takes to catch a shape of a given size. The user will probably have more difficulty catching faster moving, smaller shapes.

18.33  *(Calendar/Tickler File)* Using both audio and images create a general purpose calendar and “tickler” file. For example, the program should sing “Happy Birthday” when you use it on your birthday. Have the program display images and play audios associated with important events. Also, have the program remind you in advance of these important events. It would be nice, for example, to have the program give you a week’s notice so you can pick up an appropriate greeting card for that special person.

18.34  *(Rotating Images)* Create a Java program that lets you rotate an image through some number of degrees (out of a maximum of 360 degrees). The program should let you specify that you want to spin the image continuously. The program should let you adjust the spin speed dynamically.

18.35  *(Coloring Black and White Photographs and Images)* Create a Java program that lets you paint a black and white photograph with color. Provide a color palette for selecting colors. Your program should let you apply different colors to different regions of the image.

18.36  *(Multimedia-Based Simpletron Simulator)* Modify the Simpletron simulator that you developed in the exercises in the previous chapters to include multimedia features. Add computer-like sounds to indicate that the Simpletron is executing instructions. Add a breaking glass sound when a fatal error occurs. Use flashing lights to indicate which cells of memory and/or which registers are currently being manipulated. Use other multimedia techniques, as appropriate, to make your Simpletron simulator more valuable to its users as an educational tool.
Data Structures

Objectives

• To be able to form linked data structures using references, self-referential classes and recursion.
• To be able to create and manipulate dynamic data structures, such as linked lists, queues, stacks and binary trees.
• To understand various important applications of linked data structures.
• To understand how to create reusable data structures with classes, inheritance and composition.

Much that I bound, I could not free;
Much that I freed returned to me.
Lee Wilson Dodd

‘Will you walk a little faster?’ said a whiting to a snail,
‘There’s a porpoise close behind us, and he’s treading on my tail.’
Lewis Carroll

There is always room at the top.
Daniel Webster

Push on—keep moving.
Thomas Morton

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

We have studied such fixed-size data structures as single- and double-subscripted arrays. This chapter introduces dynamic data structures that grow and shrink at execution time. Linked lists are collections of data items “lined up in a row”—insertions and deletions can be made anywhere in a linked list. Stacks are important in compilers and operating systems; insertions and deletions are made only at one end of a stack—its top. Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head) of a queue. Binary trees facilitate high-speed searching and sorting of data, eliminating of duplicate data items efficiently, representing file system directories, compiling expressions into machine language and many other interesting applications.

We will discuss each of the major types of data structures and implement programs that create and manipulate them. We use classes, inheritance and composition to create and package these data structures for reusability and maintainability. In Chapter 20, “Java Utilities Package and Bit Manipulation,” and Chapter 21, “Collections,” we discuss Java’s pre-defined classes that implement the data structures discussed in this chapter.

The chapter examples are practical programs that can be used in more advanced courses and in industrial applications. The exercises include a rich collection of useful applications.

We encourage you to attempt the major project described in the special section entitled Building Your Own Compiler. You have been using a Java compiler to translate your Java programs to bytecodes so that you could execute these programs on your computer. In this project, you will actually build your own compiler. It will read a file of statements written in a simple, yet powerful high-level language similar to early versions of the popular language Basic. Your compiler will translate these statements into a file of Simpletron Machine Language (SML) instructions—SML is the language you learned in the Chapter 7 special section, Building Your Own Computer. Your Simpletron Simulator program will then execute the SML program produced by your compiler! Implementing this project by using an object-oriented approach will give you a wonderful opportunity to exercise most of what you have learned in this book. The special section carefully walks you through the specifications of the high-level language and describes the algorithms you will need to con-

---

### Outline

| 19.1 | Introduction |
| 19.2 | Self-Referential Classes |
| 19.3 | Dynamic Memory Allocation |
| 19.4 | Linked Lists |
| 19.5 | Stacks |
| 19.6 | Queues |
| 19.7 | Trees |

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises • Special Section: Building Your Own Compiler
vert each type of high-level language statement into machine language instructions. If you enjoy being challenged, you might attempt the many enhancements to both the compiler and the Simpletron Simulator suggested in the exercises.

19.2 Self-Referential Classes

A *self-referential class* contains an instance variable that refers to another object of the same class type. For example, the definition

```java
class Node {
    private int data;
    private Node nextNode;

    public Node( int data ) { /* constructor body */ }
    public void setData( int data ) { /* method body */ }
    public int getData() { /* method body */ }
    public void setNext( Node next ) { /* method body */ }
    public Node getNext() { /* method body */ }
}
```

defines class `Node`. This type has two `private` instance variables—integer `data` and `Node` reference `nextNode`. Member `nextNode` references an object of type `Node`, an object of the same type as the one being declared here—hence, the term “self-referential class.” Member `nextNode` is a *link*—`nextNode` “links” an object of type `Node` to another object of the same type. Type `Node` also has five methods: a constructor that receives an integer to initialize `data`, a `setData` method to set the value `data`, a `getData` method to return the value of `data`, a `setNext` method to set the value of `nextNode` and a `getNext` method to return the value of member `nextNode`.

Programs can link self-referential objects together to form such useful data structures as lists, queues, stacks and trees. Figure 19.1 illustrates two self-referential objects linked together to form a list. A backslash—representing a `null` reference—is placed in the link member of the second self-referential object to indicate that the link does not refer to another object. The backslash is for illustration purposes; it does not correspond to the backslash character in Java. Normally, a `null` reference indicates the end of a data structure.

**Common Programming Error 19.1**

Not setting the link in the last node of a list to `null` is a logic error.

19.3 Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires *dynamic memory allocation*—the ability for a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed. As we have already learned, Java programs do not explicitly release dynamically allocated memory. Rather, Java performs automatic garbage collection on objects that are no longer referenced in a program.

The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available disk space in a virtual-memory system. Often, the limits are much smaller, because the computer’s available memory must be shared among many applications.
Operator `new` is essential to dynamic memory allocation. Operator `new` takes as an operand the type of the object being dynamically allocated and returns a reference to a newly created object of that type. For example, the statement

```java
Node nodeToAdd = new Node( 10 );
```

allocates the appropriate amount of memory to store a `Node` object and places a reference to this object in `nodeToAdd`. If no memory is available, `new` throws an `OutOfMemoryError`. The 10 is the `Node` object’s data.

The following sections discuss lists, stacks, queues and trees that use dynamic memory allocation and self-referential classes to create dynamic data structures.

### 19.4 Linked Lists

A **linked list** is a linear collection (i.e., a sequence) of self-referential class objects, called **nodes**, connected by reference **links**—hence, the term “linked” list. A program accesses a linked list via a reference to the first node of the list. The program accesses each subsequent node via the link reference stored in the previous node. By convention, the link reference in the last node of a list is set to `null` to mark the end of the list. Data are stored in a linked list dynamically—the list creates each node as necessary. A node can contain data of any type, including objects of other classes. Stacks and queues are also linear data structures and, as we will see, are constrained versions of linked lists. Trees are nonlinear data structures.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary. The size of a “conventional” Java array, however, cannot be altered, because the array size is fixed at the time the program creates the array. “Conventional” arrays can become full. Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests. Package `java.util` contains class `LinkedList` for implementing and manipulating linked lists that grow and shrink during program execution. We discuss class `LinkedList` in Chapter 21, “Collections.”

**Performance Tip 19.1**

An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime.

**Performance Tip 19.2**

Insertion into a linked list is fast—only two references have to be modified (after you have located the place to do the insertion). All existing nodes remain at their current locations in memory.
Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list (it does, of course, take time to locate the proper insertion point). Existing list elements do not need to be moved.

Linked list nodes normally are not stored contiguously in memory. Rather, they are logically contiguous. Figure 19.2 illustrates a linked list with several nodes. This diagram presents a *singly-linked list*—each node contains one reference to the next node in the list. Often, linked lists are implemented as doubly-linked lists—each node contains a reference to the next node in the list and a reference to the previous node in the list. Java’s `LinkedList` class (see Chapter 21) is a doubly-linked list implementation.

---

**Performance Tip 19.3**

*Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.*

**Performance Tip 19.4**

*The elements of an array are stored contiguously in memory. This allows immediate access to any array element, because the address of any element can be calculated directly as its offset from the beginning of the array. Linked lists do not afford such immediate access to their elements—an element can be accessed only by traversing the list from the front.*

**Performance Tip 19.5**

*Using dynamic memory allocation (instead of arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that references occupy space, and that dynamic memory allocation incurs the overhead of method calls.*

The program of Fig. 19.3–Fig. 19.5 uses a `List` class to manipulate a list of miscellaneous object types. The `main` method of class `ListTest` (Fig. 19.5) creates a list of objects, inserts objects at the beginning of the list using method `insertAtFront`, inserts objects at the end of the list using method `insertAtBack`, deletes objects from the front of the list using method `removeFromFront` and deletes objects from the end of the list using method `removeFromBack`. After each insert and remove operation, `ListTest` calls `List` method `print` to display the current list contents. A detailed discussion of the program follows. If an attempt is made to remove an item from an empty list, an `EmptyListException` (Fig. 19.4) occurs.

The program of Fig. 19.3–Fig. 19.5 consists of four classes—`ListNode` (Fig. 19.3), `List` (Fig. 19.3), `EmptyListException` (Fig. 19.4) and `ListTest` (Fig. 19.5). The `List` and `ListNode` classes are placed in package `com.deitel.jhtp4.ch19`, so they can be reused throughout this chapter. Encapsulated in each `List` object is a linked list of `ListNode` objects. The `ListNode` class (lines 6–38 of Fig. 19.3) consists of package-access members `data` and `nextNode`. `ListNode` member `data` can refer to any `Object`. `ListNode` member `nextNode` stores a reference to the next `ListNode` object in the linked list.

[Note: Many of the classes in this chapter are defined in the package `com.deitel.jhtp4.ch19`. Each such class should be compiled with the `-d` command-line option to `javac`. When compiling the classes that are not in this package and when running the programs, be sure to use the `-classpath` option to `javac` and `java`, respectively.]
Fig. 19.3: List.java

// Class ListNode and class List definitions
package com.deitel.jhtp4.ch19;

// class to represent one node in a list
class ListNode {
  // package access members; List can access these directly
  Object data;
  ListNode nextNode;

  // constructor to create a ListNode that refers to object
  ListNode( Object object )
  {
    this( object, null );
  }

  // constructor to create ListNode that refers to Object
  // and to next ListNode in List
  ListNode( Object object, ListNode node )
  {
    data = object;
    nextNode = node;
  }

  // return Object in this node
  Object getObject()
  {
    return data;
  }

  // get next node
  ListNode getNext()
  {
    return nextNode;
  }
}

// end class ListNode

Fig. 19.2 A graphical representation of a linked list.

Fig. 19.3 Definitions of class ListNode and class List (part 1 of 4).
// class List definition
public class List {
    private ListNode firstNode;
    private ListNode lastNode;
    private String name;  // String like "list" used in printing

    // construct an empty List with a name
    public List( String string )
    {
        name = string;
        firstNode = lastNode = null;
    }

    // construct empty List with "list" as the name
    public List()
    {
        this( "list" );
    }

    // Insert Object at front of List. If List is empty,
    // firstNode and lastNode will refer to same object.
    // Otherwise, firstNode refers to new node.
    public synchronized void insertAtFront( Object insertItem )
    {
        if ( isEmpty() )
            firstNode = lastNode = new ListNode( insertItem );
        else
            firstNode = new ListNode( insertItem, firstNode );
    }

    // Insert Object at end of List. If List is empty,
    // firstNode and lastNode will refer to same Object.
    // Otherwise, lastNode's nextNode refers to new node.
    public synchronized void insertAtBack( Object insertItem )
    {
        if ( isEmpty() )
            firstNode = lastNode = new ListNode( insertItem );
        else
            lastNode = lastNode.nextNode =
            new ListNode( insertItem );
    }

    // remove first node from List
    public synchronized Object removeFromFront() throws EmptyListException
    {
        Object removeItem = null;
        // throw exception if List is empty
        if ( isEmpty() )
            throw new EmptyListException( name );
    }

Fig. 19.3 Definitions of class ListNode and class List (part 2 of 4).
// retrieve data being removed
removeItem = firstNode.data;

// reset the firstNode and lastNode references
if ( firstNode == lastNode )
    firstNode = lastNode = null;
else
    firstNode = firstNode.nextNode;

// return removed node data
return removeItem;

// Remove last node from List
public synchronized Object removeFromBack()
    throws EmptyListException
{
    Object removeItem = null;

    // throw exception if List is empty
    if ( isEmpty() )
        throw new EmptyListException( name );

    // retrieve data being removed
    removeItem = lastNode.data;

    // reset firstNode and lastNode references
    if ( firstNode == lastNode )
        firstNode = lastNode = null;
    else {

        // locate new last node
        ListNode current = firstNode;

        // loop while current node does not refer to lastNode
        while ( current.nextNode != lastNode )
            current = current.nextNode;

        // current is new lastNode
        lastNode = current;
        current.nextNode = null;
    }

    // return removed node data
    return removeItem;
}

// return true if List is empty
public synchronized boolean isEmpty()
{

Fig. 19.3 Definitions of class ListNode and class List (part 3 of 4).
Class `List` contains private members `firstNode` (a reference to the first `ListNode` in a `List`) and `lastNode` (a reference to the last `ListNode` in a `List`). The constructors (lines 47–51 and 54–57) initialize both references to `null`. The most important methods of class `List` are the synchronized methods `insertAtFront` (lines 62–69), `insertAtBack` (lines 74–82), `removeFromFront` (lines 85–106) and `removeFromBack` (lines 109–141). These methods are declared synchronized so `List` objects can be multithread safe when used in a multithreaded program. If one thread is modifying the contents of a `List`, no other thread can modify the same `List` object at the same time. Method `isEmpty` (lines 144–147) is a predicate method that determines whether the list is empty (i.e., the reference to the first node of the list is `null`). Predicate methods typically test a condition and do not modify the object on which they are called. If the list is empty, method `isEmpty` returns `true`; otherwise, it returns `false`. Method `print` (lines 150–168) displays the list’s contents. Both `isEmpty` and `print` are also synchronized methods.

```java
public synchronized void print() {
    if (isEmpty()) {
        System.out.println("Empty " + name);
        return;
    }
    System.out.print("The " + name + " is: ");
    ListNode current = firstNode;
    while (current != null) {
        System.out.print(current.data.toString() + " ");
        current = current.nextNode;
    }
    System.out.println("\n");
}
```

Fig. 19.3 Definitions of class `ListNode` and class `List` (part 4 of 4).

Fig. 19.4 Definition of class `EmptyListException` (part 1 of 2).
// initialize an EmptyListException
public EmptyListException( String name )
{
    super( "The " + name + " is empty" );
}
}  // end class EmptyListException

// Fig. 19.5: ListTest.java
// Class ListTest
// Deitel packages
import com.deitel.jhtp4.ch19.List;
import com.deitel.jhtp4.ch19.EmptyListException;

public class ListTest {

    // test class List
    public static void main( String args[] )
    {
        List list = new List();  // create the List container
        // create objects to store in List
        Boolean bool = Boolean.TRUE;
        Character character = new Character( '$' );
        Integer integer = new Integer( 34567 );
        String string = "hello";

        // use List insert methods
        list.insertAtFront( bool );
        list.print();
        list.insertAtFront( character );
        list.print();
        list.insertAtBack( integer );
        list.print();
        list.insertAtBack( string );
        list.print();

        // use List remove methods
        Object removedObject;

        // remove objects from list; print after each removal
        try {
            removedObject = list.removeFromFront();
            System.out.println(removedObject.toString() + " removed");
            list.print();
        }
    }
}  // end class ListTest

Fig. 19.5 Manipulating a linked list.
over the next several pages, we discuss each of the methods of class List (Fig. 19.3) in detail. Method insertAtFront (lines 62–69 of Fig. 19.3) places a new node at the front of the list. The method consists of several steps (Fig. 19.6 illustrates the operation):

1. Call isEmpty to determine whether the list is empty (line 64).

```java
removedObject = list.removeFromFront();
System.out.println(
    removedObject.toString() + " removed" );
list.print();

removedObject = list.removeFromBack();
System.out.println(
    removedObject.toString() + " removed" );
list.print();

removedObject = list.removeFromBack();
System.out.println(
    removedObject.toString() + " removed" );
list.print();
```

// process exception if List is empty when attempt is made to remove an item
```java
catch ( EmptyListException emptyListException ) {
    emptyListException.printStackTrace();
}
```

```java
// end method main
```

```java
// end class ListTest
```

The list is: true

The list is: $ true

The list is: $ true 34567

The list is: $ true 34567 hello

$ removed
The list is: true 34567 hello

ture removed
The list is: 34567 hello

hello removed
The list is: 34567

34567 removed
Empty list

Fig. 19.5 Manipulating a linked list.

Over the next several pages, we discuss each of the methods of class List (Fig. 19.3) in detail. Method insertAtFront (lines 62–69 of Fig. 19.3) places a new node at the front of the list. The method consists of several steps (Fig. 19.6 illustrates the operation):

1. Call isEmpty to determine whether the list is empty (line 64).
2. If the list is empty, both `firstNode` and `lastNode` are set to the `ListNode` allocated with `new` and initialized with `insertItem` (line 65). The `ListNode` constructor at lines 13–16 calls the `ListNode` constructor at lines 20–24 to set instance variable `data` to refer to the `insertItem` passed as an argument and to set reference `nextNode` to `null`.

3. If the list is not empty, the new node is “threaded” (not to be confused with multi-threading) or “linked” into the list by setting `firstNode` to a new `ListNode` object and initializing that object with `insertItem` and `firstNode` (line 68). When the `ListNode` constructor (lines 20–24) executes, it sets instance variable `data` to refer to the `insertItem` passed as an argument and performs the insertion by setting the `nextNode` reference to the `ListNode` passed as an argument.

In Fig. 19.6, part (a) shows the list and the new node during the `insertAtFront` operation and before the program threads the new node into the list. The dotted arrows in part (b) illustrate step 3 of the `insertAtFront` operation that enables the node containing 12 to become the new list front.

Method `insertAtBack` (lines 74–82 of Fig. 19.3) places a new node at the back of the list. The method consists of several steps (Fig. 19.7 illustrates the operation):

1. Call `isEmpty` to determine whether the list is empty (line 76).

2. If the list is empty, both `firstNode` and `lastNode` are set to the `ListNode` allocated with `new` and initialized with `insertItem` (line 77). The `ListNode` constructor at lines 13–16 calls the `ListNode` constructor at lines 20–24 to set instance variable `data` to refer to the `insertItem` passed as an argument and to set reference `nextNode` to `null`.

3. If the list is not empty, the new node is threaded into the list by setting `LastNode` and `lastNode.nextNode` to the `ListNode` that was allocated with `new` and initialized with `insertItem` (lines 80–81). When the `ListNode` constructor (lines 13–16) executes, it sets instance variable `data` to refer to the `insertItem` passed as an argument and sets reference `nextNode` to `null`.

![Fig. 19.6](image_url)
In Fig. 19.7, part (a) shows the list and the new node during the `insertAtBack` operation and before the program threads the new node into the list. The dotted arrows in part (b) illustrate the steps of method `insertAtBack` that enable a new node to be added to the end of a list that is not empty.

Method `removeFromFront` (lines 85–106 of Fig. 19.3) removes the front node of the list and returns a reference to the removed data. The method throws an `EmptyList-Exception` (lines 91–92) if the list is empty when the program calls this method. Otherwise, the method returns a reference to the removed data. The method consists of several steps (illustrated in Fig. 19.8):

1. Assign `firstNode.data` (the data being removed from the list) to reference `removeItem` (line 95).
2. If the objects to which `firstNode` and `lastNode` refer are equal (line 98), the list has only one element prior to the removal attempt. In this case, the method sets `firstNode` and `lastNode` to null (line 99) to “dethread” (remove) the node from the list (leaving the list empty).
3. If the list has more than one node prior to removal, then the method leaves reference `lastNode` as is and simply assigns `firstNode.nextNode` to reference `firstNode` (line 102). Thus, `firstNode` references the node that was the second node prior to the `removeFromFront` call.
4. Return the `removeItem` reference (line 105).

In Fig. 19.8, part (a) illustrates the list before the removal operation. Part (b) shows actual reference manipulations.
Method `removeFromBack` (lines 109–141 of Fig. 19.3) removes the last node of a list and returns a reference to the removed data. The method throws an `EmptyListException` (lines 94 and 95) if the list is empty when the program calls this method. The method consists of several steps (Fig. 19.9 illustrates the operation):

1. Assign `lastNode.data` (the data being removed from the list) to reference `removeItem` (line 119).

2. If the objects to which `firstNode` and `lastNode` refer are equal (line 122), the list has only one element prior to the removal attempt. In this case, the method sets `firstNode` and `lastNode` to null (line 123) to dethread (remove) that node from the list (leaving the list empty).

3. If the list has more than one node prior to removal, then create the `ListNode` reference `current` and assign it `firstNode`.

4. Now “walk the list” with `current` until it references the node before the last node. The while loop (lines 131–132) assigns `current.nextNode` to `current` as long as `current.nextNode` is not `lastNode`.

5. After locating the second-to-last node, assign `current` to `lastNode` (line 135) to dethread the last node from the list.

6. Set the `current.nextNode` to null (line 136) to terminate the list at the current node.

7. Return the `removeItem` reference (liner 140).

In Fig. 19.9, part (a) illustrates the list before the removal operation. Part (b) shows the actual reference manipulations.
1108 Data Structures Chapter 19

Method `print` (lines 150–168) first determines whether the list is empty (lines 152–155). If so, `print` displays a message indicating that the list is empty and returns control to the calling method. Otherwise, `print` outputs the data in the list. Line 159 creates `ListNode` reference `current` and initializes it with `firstNode`. While `current` is not `null`, there are more items in the list. Therefore, line 163 outputs a `String` representation of `current.data`. Line 164 moves to the next node in the list by assigning the value of `current.nextNode` to `current`. This printing algorithm is identical for linked lists, stacks and queues.

19.5 Stacks

A stack is a constrained version of a linked list—new nodes can be added to a stack and removed from a stack only at the top. For this reason, a stack is referred to as a *last-in, first-out* (LIFO) data structure. The link member in the bottom (i.e., last) node of the stack is set to `null` to indicate the bottom of the stack.

**Common Programming Error 19.2**

Not setting the link in the bottom node of a stack to `null` is a common logic error.

The primary methods used to manipulate a stack are `push` and `pop`. Method `push` adds a new node to the top of the stack. Method `pop` removes a node from the top of the stack and returns the data from the popped node.

Stacks have many interesting applications. For example, when a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the *program execution stack*. If a series of method calls
occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that each method can return to its caller. Stacks support recursive method calls in the same manner as they do conventional nonrecursive method calls.

The program execution stack contains the memory for local variables on each invocation of a method during a program’s execution. When the method returns to its caller, the memory for that method’s local variables is popped off the stack and those variables are no longer known to the program.

Compilers use stacks to evaluate arithmetic expressions and generate machine language code to process the expressions. The exercises in this chapter explore several applications of stacks, including using them to develop a complete working compiler. The java.util package of the Java API contains class Stack for implementing and manipulating stacks that can grow and shrink during program execution. We will discuss class Stack in Chapter 20.

We take advantage of the close relationship between lists and stacks to implement a stack class by reusing a list class. We demonstrate two different forms of reusability. First, we implement the stack class by extending class List of Fig. 19.3. Then we implement an identically performing stack class through composition by including a List object as a private member of a stack class. The list, stack and queue data structures in this chapter are implemented to store Object references to encourage further reusability. Thus, any object type can be stored in a list, stack or queue.

The application of Fig. 19.10 and Fig. 19.11 creates a stack class by extending class List of Fig. 19.3. We want the stack to have methods push, pop, isEmpty and print. Essentially, these are the methods insertAtFront, removeFromFront, isEmpty and print of class List. Of course, class List contains other methods (such as insertAtBack and removeFromBack) that we would rather not make accessible through the public interface to the stack class. It is important to remember that all methods in the public interface of class List class also are public methods of the sub-class StackInheritance (Fig. 19.10). When we implement the stack’s methods, we have each StackInheritance method call the appropriate List method—method push calls insertAtFront and method pop calls removeFromFront. Class StackInheritance is defined as part of package com.deitel.jhtp4.ch19 for reuse purposes. Note that StackInheritance does not import List, because both classes are in the same package.

```java
// Fig. 19.10: StackInheritance.java
// Derived from class List
package com.deitel.jhtp4.ch19;

public class StackInheritance extends List {

    // construct stack
    public StackInheritance()
    {
        super( "stack" );
    }
}
```

Fig. 19.10 Class StackInheritance extends class List (part 1 of 2).
Class `StackInheritanceTest`’s method `main` (Fig. 19.11) uses class `StackInheritance` to instantiate a stack of `Objects` called `stack`. The program pushes onto the stack (lines 22, 24, 26 and 28) a `Boolean` object containing `true`, a `Character` object containing `$`, an `Integer` object containing `34567` and a `String` object containing `hello`, then popped off `stack`. Lines 37–42 pop the objects from the stack in an infinite `while` loop. When there are no objects left to pop, an method `pop` throws an `EmptyListException`, and the program displays the exception’s stack trace, which shows the methods on the program execution stack at the time the exception occurred. Note that the program uses method `print` (inherited from `List`) to output the contents of the stack.

```java
13     // add object to stack
14     public synchronized void push( Object object )
15     {
16         insertAtFront( object );
17     }
18
19     // remove object from stack
20     public synchronized Object pop() throws EmptyListException
21     {
22         return removeFromFront();
23     }
24 } // end class StackInheritance

Fig. 19.10 Class `StackInheritance` extends class `List` (part 2 of 2).

Fig. 19.11 A simple stack program (part 1 of 2).
```java
stack.print();
stack.push( integer );
stack.print();
stack.push( string );
stack.print();

// remove items from stack
try {
    // use pop method
    Object removedObject = null;
    while ( true ) {
        removedObject = stack.pop();
        System.out.println( removedObject.toString() + " popped" );
        stack.print();
    }
}

// catch exception if stack empty when item popped
catch ( EmptyListException emptyListException ) {
    emptyListException.printStackTrace();
}

} // end method main

// end class StackInheritanceTest
```

The stack is: true
The stack is: $ true
The stack is: 34567 $ true
The stack is: hello 34567 $ true
hello popped
The stack is: 34567 $ true
34567 popped
The stack is: $ true
$ popped
The stack is: true
true popped
Empty stack
com.deitel.jhtp4.ch19.EmptyListException: The stack is empty
    at com.deitel.jhtp4.ch19.List.removeFromFront(List.java:92)
    at com.deitel.jhtp4.ch19.StackInheritance.pop(StackInheritance.java:22)
    at StackInheritanceTest.main(StackInheritanceTest.java:38)

Fig. 19.11 A simple stack program (part 2 of 2).
Another way to implement a stack class is by reusing a list class through composition. The class in Fig. 19.12 uses a \texttt{private} object of class \texttt{List} (line 6) in the definition of class \texttt{StackComposition}. Composition enables us to hide the methods of class \texttt{List} that should not be in our stack’s \texttt{public} interface by providing \texttt{public} interface methods only to the required \texttt{List} methods. This technique of implementing each stack method as a call to a \texttt{List} method is called \textit{delegating}—the stack method invoked \textit{delegates} the call to the appropriate \texttt{List} method. In particular, \texttt{StackComposition} delegates calls to \texttt{List} methods \texttt{insertAtFront}, \texttt{removeFromFront}, \texttt{isEmpty} and \texttt{print}. In this example, we do not show class \texttt{StackCompositionTest}, because the only difference in this example is that we change the type of the stack from \texttt{StackInheritance} to \texttt{StackComposition}. If you execute the application from the code on the CD that accompanies this book, you will see that the output is identical.

```java
package com.deitel.jhtp4.ch19;

public class StackComposition {

    private List stackList;

    public StackComposition() {
        stackList = new List( "stack" );
    }

    public synchronized void push( Object object ) {
        stackList.insertAtFront( object );
    }

    public synchronized Object pop() throws EmptyListException {
        return stackList.removeFromFront();
    }

    public synchronized boolean isEmpty() {
        return stackList.isEmpty();
    }

    public synchronized void print() {
        stackList.print();
    }

} // end class StackComposition
```

Fig. 19.12 A simple stack class using composition.
19.6 Queues

Another common data structure is the queue. A queue is similar to a checkout line in a supermarket—the cashier services the person at the beginning of the line first. Other customers enter the line only at the end and wait for service. Queue nodes are removed only from the head (or front) of the queue and are inserted only at the tail (or end) of the queue. For this reason, a queue is a first-in, first-out (FIFO) data structure. The insert and remove operations are known as enqueue and dequeue.

Queues have many applications in computer systems. Most computers have only a single processor, so only one application at a time can be serviced. Entries for the other applications are placed in a queue. The entry at the front of the queue is the next to receive service. Each entry gradually advances to the front of the queue as applications receive service.

Queues are also used to support print spooling. A multiuser environment may have only a single printer. Many users may send print jobs to the printer. Even when the printer is busy, other people may still send print jobs. These are “spooled” to disk (much as thread is wound onto a spool) where they wait in a queue until the printer becomes available.

Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet’s final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

A file server in a computer network handles file-access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

The application of Fig. 19.13 and Fig. 19.14 creates a queue by extending class List (Fig. 19.3). We want class QueueInheritance (Fig. 19.13) to have methods enqueue and dequeue.isEmpty and print. Note that these are essentially the List methods insertAtBack and removeFromFront. Of course, class List contains other methods (i.e., insertAtFront and removeFromBack) that we would rather not make accessible through the public interface of the queue. Remember that all methods in the public interface of the List class are also public methods of the subclass QueueInheritance. In the queue’s implementation, each method of class QueueInheritance calls the appropriate List method—method enqueue calls insertAtBack and method dequeue calls removeFromFront. Class QueueInheritance is defined in package com.deitel.jhtp4.ch19 for reuse purposes.

Common Programming Error 19.3

Not setting the link in the last node of a queue to null is a common logic error.
Class QueueInheritanceTest method main (Fig. 19.14) uses class QueueInheritance to instantiate a queue of Objects called queue. Lines 22, 24, 26 and 28 enqueue a Boolean object containing true, a Character object containing $, an Integer object containing 34567 and a String object containing hello. Lines 37–42 use an infinite while loop to dequeue the objects in first-in, first-out order. When there are no objects left to dequeue, method dequeue throws an EmptyListException, and the program displays the exception’s stack trace.

```java
// Fig. 19.14: QueueInheritanceTest.java
// Class QueueInheritanceTest

public class QueueInheritanceTest {
    public static void main( String args[] )
    {
        QueueInheritance queue = new QueueInheritance();
        // create objects to store in queue
        Boolean bool = Boolean.TRUE;
        Character character = new Character( '$' );
        Integer integer = new Integer( 34567 );
        String string = "hello";
```

---

**Fig. 19.13** Class QueueInheritance extends class List (part 2 of 2).

**Fig. 19.14** Processing a queue (part 1 of 3).
// use enqueue method
queue.enqueue( bool );
queue.print();
queue.enqueue( character );
queue.print();
queue.enqueue( integer );
queue.print();
queue.enqueue( string );
queue.print();

// remove objects from queue
try {
    // use dequeue method
    Object removedObject = null;
    while ( true ) {
        removedObject = queue.dequeue();
        System.out.println( removedObject.toString() + " dequeued" );
        queue.print();
    }
}

// process exception if queue empty when item removed
catch ( EmptyListException emptyListException ) {
    emptyListException.printStackTrace();
}

} // end method main

} // end class QueueInheritanceTest

The queue is: true
The queue is: true $
The queue is: true $ 34567
The queue is: true $ 34567 hello
true dequeued
The queue is: $ 34567 hello
$ dequeued
The queue is: 34567 hello
34567 dequeued
The queue is: hello
hello dequeued
Empty queue

(continued on next page)
19.7 Trees

Linked lists, stacks and queues are linear data structures (i.e., sequences). A tree is a non-linear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses binary trees (Fig. 19.15)—trees whose nodes all contain two links (none, one or both of which may be null). The root node is the first node in a tree. Each link in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree. The children of a specific node are called siblings. A node with no children is called a leaf node. Computer scientists normally draw trees from the root node down—exactly the opposite of the way most trees grow in nature.

Common Programming Error 19.4

Not setting to null the links in leaf nodes of a tree is a common logic error.

In our binary tree example, we create a special binary tree called a binary search tree. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in the subtree’s parent node, and the values in any right subtree are greater than the value in the subtree’s parent node. Figure 19.16 illustrates a binary search tree with 12 integer values. Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

Fig. 19.14 Processing a queue (part 3 of 3).

(continued from previous page)

```
com.deitel.jhtp4.ch19.EmptyListException: The queue is empty
    at com.deitel.jhtp4.ch19.List.removeFromFront(List.java:92)
    at com.deitel.jhtp4.ch19.QueueInheritance.dequeue(QueueInheritance.java:24)
    at QueueInheritanceTest.main(QueueInheritanceTest.java:38)
```

Fig. 19.14 Processing a queue (part 3 of 3).

Fig. 19.15 A graphical representation of a binary tree.
The application of Fig. 19.17 and Fig. 19.18 creates a binary search tree of integers and traverses it (i.e., walks through all its nodes) three ways—using recursive *inorder*, *pre-order* and *postorder* traversals. The program generates 10 random numbers and inserts each into the tree. Class `Tree` is defined in package `com.deitel.jhtp4.ch19` for reuse purposes.

```java
// Fig. 19.17: Tree.java
// Definition of class TreeNode and class Tree.

// Deitel packages
package com.deitel.jhtp4.ch19;

// class TreeNode definition
class TreeNode {

    // package access members
    TreeNode leftNode;
    int data;
    TreeNode rightNode;

    // initialize data and make this a leaf node
    public TreeNode( int nodeData ) {
        data = nodeData;
        leftNode = rightNode = null; // node has no children
    }

    // insert TreeNode into Tree that contains nodes;
    // ignore duplicate values
    public synchronized void insert( int insertValue ) {
        // insert in left subtree
        if ( insertValue < data ) {
            if ( leftNode == null )
                leftNode = new TreeNode( insertValue );
        }
    }

    // print all nodes in a level order (breadth-first) manner
    public synchronized void levelOrder() {
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add( this );
        while ( !queue.isEmpty() ) {
            TreeNode current = queue.poll();
            System.out.println( current.data );
            if ( current.leftNode != null )
                queue.add( current.leftNode );
            if ( current.rightNode != null )
                queue.add( current.rightNode );
        }
    }

    // print all nodes in a leaf to root order (depth-first) manner
    public synchronized void leafToRootOrder() {
        if ( leftNode != null )
            leafToRootOrder( leftNode );
        if ( rightNode != null )
            leafToRootOrder( rightNode );
        System.out.println( data );
    }

    // print all nodes in a root to leaf order (depth-first) manner
    public synchronized void rootToLeafOrder() {
        System.out.println( data );
        if ( leftNode != null )
            rootToLeafOrder( leftNode );
        if ( rightNode != null )
            rootToLeafOrder( rightNode );
    }
}
```

**Fig. 19.17** Definitions of `TreeNode` and `Tree` for a binary search tree (part 1 of 4).
33     // continue traversing left subtree
34     else
35         leftNode.insert( insertValue );
36     }
37
38     // insert in right subtree
39     else if ( insertValue > data ) {
40         // insert new TreeNode
41         if ( rightNode == null )
42             rightNode = new TreeNode( insertValue );
43         // continue traversing right subtree
44         else
45             rightNode.insert( insertValue );
46     }
47
48 }  // end method insert
49
50 }  // end class TreeNode
51
52 // class Tree definition
53 public class Tree {  
54     private TreeNode root;
55 }
56
57 // construct an empty Tree of integers
58 public Tree()
59 {
60     root = null;
61 }
62
63 // Insert a new node in the binary search tree.
64 // If the root node is null, create the root node here.
65 // Otherwise, call the insert method of class TreeNode.
66 public synchronized void insertNode( int insertValue )
67 {
68     if ( root == null )
69         root = new TreeNode( insertValue );
70     else
71         root.insert( insertValue );
72 }
73
74 // begin preorder traversal
75 public synchronized void preorderTraversal()
76 {
77     preorderHelper( root );
78 }
79
80 // recursive method to perform preorder traversal
81 private void preorderHelper( TreeNode node )
82 {
83     // Fig. 19.17 Definitions of TreeNode and Tree for a binary search tree (part 2 of 4).
if ( node == null )
    return;

// output node data
System.out.print( node.data + " " );

// traverse left subtree
preorderHelper( node.leftNode );

// traverse right subtree
preorderHelper( node.rightNode );

// begin inorder traversal
public synchronized void inorderTraversal()
{
    inorderHelper( root );
}

// recursive method to perform inorder traversal
private void inorderHelper( TreeNode node )
{
    if ( node == null )
        return;

    // traverse left subtree
    inorderHelper( node.leftNode );

    // output node data
    System.out.print( node.data + " " );

    // traverse right subtree
    inorderHelper( node.rightNode );

    // begin postorder traversal
    public synchronized void postorderTraversal()
    {
        postorderHelper( root );
    }

    // recursive method to perform postorder traversal
    private void postorderHelper( TreeNode node )
    {
        if ( node == null )
            return;

        // traverse left subtree
        postorderHelper( node.leftNode );

        // output node data
        System.out.print( node.data + " " );

        // traverse right subtree
        postorderHelper( node.rightNode );

        // Definitions of TreeNode and Tree for a binary search tree (part 3 of 4).
Let us walk through the binary tree program. Method `main` of class `TreeTest` (Fig. 19.18) begins by instantiating an empty `Tree` object and storing it in reference `tree` (line 11). The program randomly generates 10 integers, each of which is inserted into the binary tree through a call to `synchronized` method `insertNode` (line 21). The program then performs preorder, inorder and postorder traversals (these will be explained shortly) of `tree`.

```java
// Fig. 19.18: TreeTest.java
// This program tests class Tree.
import com.deitel.jhtp4.ch19.Tree;

public class TreeTest {

    // test class Tree
    public static void main( String args[] )
    {
        Tree tree = new Tree();
        int value;

        System.out.println( "Inserting the following values: " );

        // insert 10 random integers from 0-99 in tree
        for ( int i = 1; i <= 10; i++ ) {
            value = ( int ) ( Math.random() * 100 );
            System.out.print( value + " " );
            tree.insertNode( value );
        }

        // perform preorder traversal of tree
        System.out.println ( "\n\nPreorder traversal" );
        tree.preorderTraversal();

        // perform inorder traversal of tree
        System.out.println ( "\n\nInorder traversal" );
        tree.inorderTraversal();

        // perform postorder traversal of tree
        System.out.println ( "\n\nPostorder traversal" );
        tree.postorderTraversal();
    }
}
```

Fig. 19.17 Definitions of `TreeNode` and `Tree` for a binary search tree (part 4 of 4).

Fig. 19.18 Creating and traversing a binary tree (part 1 of 2).
Class `Tree` (lines 55–142 of Fig. 19.17) has as private data `root`—a reference to the root node of the tree. The class contains public method `insertNode` (lines 67–74) to insert a new node in the tree and public methods `preorderTraversal` (lines 77–80), `inorderTraversal` (lines 99–102) and `postorderTraversal` (lines 121–124) to begin traversals of the tree. Each of these methods calls a separate recursive utility method to perform the traversal operations on the internal representation of the tree. The `Tree` constructor (lines 59–62) initializes `root` to `null` to indicate that the tree is empty.

The `Tree` class’s synchronized method `insertNode` (lines 67–74) first determines if the tree is empty. If so, it allocates a new `TreeNode`, initializes the node with the integer being inserted in the tree and assigns the new node to the `root` reference (line 70). If the tree is not empty, `insertNode` calls `TreeNode` method `insert` (lines 24–52). This method uses recursion to determine the location for the new node in the tree and inserts the node at that location. A node can be inserted only as a leaf node in a binary search tree.

Methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` call helper methods `inorderHelper` (lines 105–118), `preorderHelper` (lines 83–96) and `postorderHelper` (lines 127–140), respectively, to traverse the tree (Fig. 19.19) and print the node values. The purpose of the helper methods in class `Tree` is to allow the programmer to start a traversal without the need to obtain a reference to the `root` node first, then call the recursive method with that reference. Methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` simply take the private `root` reference and pass it to the appropriate helper method to initiate a traversal of the tree.

```
37 38 } // end class TreeTest

Inserting the following values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39
```

Fig. 19.18 Creating and traversing a binary tree (part 2 of 2).
Method `inorderHelper` (lines 105–118) defines the steps for an inorder traversal. Those steps are as follows:

1. Traverse the left subtree with a call to `inorderHelper` (line 111).
2. Process the value in the node (line 114).
3. Traverse the right subtree with a call to `inorderHelper` (line 117).

The inorder traversal does not process the value in a node until the values in that node’s left subtree are processed. The inorder traversal of the tree in Fig. 19.19 is

```
6 13 17 27 33 42 48
```

Note that the inorder traversal of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually sorts the data—and thus, this process is called the *binary tree sort*.

Method `preorderHelper` (lines 83–96) defines the steps for a preorder traversal. Those steps are as follows:

1. Process the value in the node (line 89).
2. Traverse the left subtree with a call to `preorderHelper` (line 92).
3. Traverse the right subtree with a call to `preorderHelper` (line 95).

The preorder traversal processes the value in each node as the node is visited. After processing the value in a given node, the preorder traversal processes the values in the left subtree, then the values in the right subtree. The preorder traversal of the tree in Fig. 19.19 is

```
27 13 6 17 42 33 48
```

Method `postorderHelper` (lines 127–140) defines the steps for a postorder traversal. Those steps are as follows:

1. Traverse the left subtree with a `postorderHelper` (line 133).
2. Traverse the right subtree with a `postorderHelper` (line 136).
3. Process the value in the node (line 139).

The postorder traversal processes the value in each node after the values of all that node’s children are processed. The *postorderTraversal* of the tree in Fig. 19.19 is

```
6 17 13 33 48 42 27
```

The binary search tree facilitates *duplicate elimination*. While building a tree, the insertion operation recognizes attempts to insert a duplicate value, because a duplicate follows
the same “go left” or “go right” decisions on each comparison as the original value did. Thus, the insertion operation eventually compares the duplicate with a node containing the same value. At this point, the insertion operation might simply discard the duplicate value.

Searching a binary tree for a value that matches a key value is fast, especially for tightly packed trees. In a tightly packed tree, each level contains about twice as many elements as the previous level. Figure 19.19 is a tightly packed binary tree. A binary search tree with \( n \) elements has a minimum of \( \log_2 n \) levels. Thus, at most \( \log_2 n \) comparisons are required either to find a match or to determine that no match exists. Searching a (tightly packed) 1000-element binary search tree requires at most 10 comparisons, because \( 2^{10} > 1000 \). Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because \( 2^{20} > 1,000,000 \).

The chapter exercises present algorithms for several other binary tree operations, such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a \textit{level-order traversal of a binary tree}. The level-order traversal of a binary tree visits the nodes of the tree row-by-row, starting at the root node level. On each level of the tree, a level-order traversal visits the nodes from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

\textbf{SUMMARY}

- Dynamic data structures can grow and shrink at execution time.
- Linked lists are collections of data items “lined up in a row”—insertions and deletions can be made anywhere in a linked list.
- Stacks are important in compilers and operating systems—insertions and deletions are made only at one end of a stack, its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, eliminating duplicate data items efficiently, representing file system directories and compiling expressions into machine language.
- A self-referential class contains a reference that refers to another object of the same class type. Self-referential objects can be linked together to form useful data structures such as lists, queues, stacks and trees.
- Creating and maintaining dynamic data structures requires dynamic memory allocation—the ability for a program to obtain more memory space at execution time to hold new nodes and to release space no longer needed.
- The limit for dynamic memory allocation can be as large as the available physical memory in the computer or the amount of available disk space in a virtual-memory system. Often, the limits are much smaller because the computer’s available memory must be shared among many users.
- Operator \texttt{new} takes as an operand the type of the object being dynamically allocated and returns a reference to a newly created object of that type. If no memory is available, \texttt{new} throws an \texttt{Out-OfMemoryError}.
- A linked list is a linear collection (i.e., a sequence) of self-referential class objects, called nodes, connected by reference links.
- A linked list is accessed via a reference to the first node of the list. Each subsequent node is accessed via the link-reference member stored in the previous node.
• By convention, the link reference in the last node of a list is set to null to mark the end of the list.

• A node can contain data of any type, including objects of other classes.

• Trees are nonlinear data structures.

• A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary.

• The size of a “conventional” Java array cannot be altered—the size is fixed at creation time.

• Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list.

• List nodes are normally not stored contiguously in memory. Rather, they are logically contiguous.

• Methods that manipulate the contents of a list should be declared synchronized so list objects can be multithread safe when used in a multithreaded program. If one thread is modifying the contents of a list, no other thread is allowed to modify the same list at the same time.

• A stack is a constrained version of a linked list—new nodes can be added to a stack and removed from a stack only at the top. A stack is referred to as a last-in, first-out (LIFO) data structure.

• The primary methods used to manipulate a stack are push and pop. Method push adds a new node to the top of the stack. Method pop removes a node from the top of the stack and returns the data object from the popped node.

• Stacks have many interesting applications. When a method call is made, the called method must know how to return to its caller, so the return address is pushed onto the program execution stack. If a series of method calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each method can return to its caller.

• The program execution stack contains the space created for local variables on each invocation of a method. When the method returns to its caller, the space for that method’s local variables is popped off the stack, and those variables are no longer known to the program.

• Stacks are also used by compilers in the process of evaluating arithmetic expressions and generating machine language code to process the expressions.

• The technique of implementing each stack method as a call to a List method is called delegating—the stack method invoked delegates the call to the appropriate List method.

• A queue is a constrained version of a list.

• A queue is similar to a checkout line in a supermarket—the first person in line is serviced first, and other customers enter the line only at the end and wait to be serviced.

• Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue. For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.

• The insert and remove operations for a queue are known as enqueue and dequeue.

• Queues have many applications in computer systems. Most computers have only a single processor, so only one user at a time can be serviced. Entries for the other users are placed in a queue. The entry at the front of the queue is the next to receive service. Each entry gradually advances to the front of the queue as users receive service.

• Queues are also used to support print spooling. A multiuser environment might have only a single printer. Many users may be generating outputs to be printed. If the printer is busy, other outputs may still be generated. These are “spooled” to disk (much as thread is wound onto a spool) where they wait in a queue until the printer becomes available.

• Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet’s final
destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

- A file server in a computer network handles file-access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.
- A tree is a nonlinear, two-dimensional data structure.
- Tree nodes contain two or more links.
- A binary tree is a tree whose nodes all contain two links. The root node is the first node in a tree.
- Each link in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree.
- The children of a node are called siblings. A node with no children is called a leaf node.
- Computer scientists normally draw trees from the root node down.
- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.
- A node can be inserted only as a leaf node in a binary search tree.
- An inorder traversal of a binary search tree processes the node values in ascending order.
- The process of creating a binary search tree actually sorts the data—and thus this process is called the binary tree sort.
- In a preorder traversal, the value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed, then the values in the right subtree are processed.
- In a postorder traversal, the value in each node is processed after the values of its children.
- The binary search tree facilitates duplicate elimination. As the tree is created, attempts to insert a duplicate value are recognized because a duplicate follows the same “go left” or “go right” decisions on each comparison as the original value did. Thus, the duplicate eventually is compared with a node containing the same value. The duplicate value could simply be discarded at this point.
- Searching a binary tree for a value that matches a key value is also fast, especially for tightly packed trees. In a tightly packed tree, each level contains about twice as many elements as the previous level. So a binary search tree with \( n \) elements has a minimum of \( \log_2 n \) levels, and thus at most \( \log_2 n \), comparisons would have to be made either to find a match or to determine that no match exists. Searching a (tightly packed) 1000-element binary search tree requires at most 10 comparisons, because \( 2^{10} > 1000 \). Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because \( 2^{20} > 1,000,000 \).

**TERMINOLOGY**

- binary search tree
- enqueue
- binary tree
- FIFO (first-in, first-out)
- binary tree sort
- head of a queue
- child node
- inorder traversal of a binary tree
- children
- inserting a node
deleating a node
- leaf node
dequeue
- left child
duplicate elimination
- left subtree
dynamic data structures
- level-order traversal of a binary tree
- LIFO (last-in, first-out)
SELF-REVIEW EXERCISES

19.1 Fill in the blanks in each of the following statements:
   a) A self-________ class is used to form dynamic data structures that can grow and shrink at execution time.
   b) Operator ________ dynamically allocates memory; this operator returns a reference to the allocated memory.
   c) A ________ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list; this data structure returns node values in last-in, first-out order.
   d) A method that does not alter a linked list, but simply looks at the list to determine whether it is empty is referred to as a ________ method.
   e) A queue is referred to as a ________ data structure because the first nodes inserted are the first nodes removed.
   f) The reference to the next node in a linked list is referred to as a ________.
   g) Automatically reclaiming dynamically allocated memory in Java is called ________.
   h) A ________ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.
   i) A ________ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
   j) A stack is referred to as a ________ data structure because the last node inserted is the first node removed.
   k) The nodes of a ________ tree contain two link members.
   l) The first node of a tree is the ________ node.
   m) Each link in a tree node refers to a ________ or ________ of that node.
   n) A tree node that has no children is called a ________ node.
   o) The four traversal algorithms we mentioned in the text for binary search trees are ________, ________, ________ and ________.

19.2 What are the differences between a linked list and a stack?

19.3 What are the differences between a stack and a queue?

19.4 Perhaps a more appropriate title for this chapter would have been “Reusable Data Structures.” Comment on how each of the following entities or concepts contributes to the reusability of data structures:
   a) classes
   b) inheritance
   c) composition
19.5 Manually provide the inorder, preorder and postorder traversals of the binary search tree of Fig. 19.20.

**ANSWERS TO SELF-REVIEW EXERCISES**

19.1 a) referential.  b) `new`.  c) stack.  d) predicate.  e) first-in, first-out (FIFO).  f) link.  
g) garbage collection.  h) queue.  i) tree  j) last-in, first-out (LIFO).  k) binary.  l) root.  m) child or subtree.  
n) leaf.  o) inorder, preorder, postorder, level order.

19.2 It is possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may only be inserted at the top of the stack and removed from the top of a stack.

19.3 A queue is a FIFO data structure that has references to both its head and its tail so that nodes may be inserted at the tail and deleted from the head. A stack is a LIFO data structure that has a single reference to the top of the stack where both insertion and deletion of nodes are performed.

19.4 a) Classes allow us to instantiate as many data structure objects of a certain type (i.e., class) as we wish.

b) Inheritance enables us to reuse code from a superclass in a subclass so that the derived class data structure is also a base-class data structure.

c) Composition enables us to reuse code by making a class object data structure a member of a composed class; if we make the class object a private member of the composed class, then the class object’s public methods are not available through the composed object’s interface.

19.5 The inorder traversal is

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

The preorder traversal is

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

The postorder traversal is

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

**EXERCISES**

19.6 Write a program that concatenates two linked-list objects of characters. Class `ListConcatenate` should include a method `concatenate` that takes references to both list objects as arguments and concatenates the second list to the first list.
19.7 Write a program that merges two ordered-list objects of integers into a single ordered list object of integers. Method `merge` of class `ListMerge` should receive references to each of the list objects to be merged and should return a reference to the merged list object.

19.8 Write a program that inserts 25 random integers from 0 to 100 in order into a linked list object. The program should calculate the sum of the elements and the floating-point average of the elements.

19.9 Write a program that creates a linked list object of 10 characters, then creates a second list object containing a copy of the first list, but in reverse order.

19.10 Write a program that inputs a line of text and uses a stack object to print the words of the line in reverse order.

19.11 Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

19.12 Stacks are used by compilers to help in the process of evaluating expressions and generating machine language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like 3 + 4 and 7 / 9 in which the operator (+ or / here) is written between its operands—this is called **infix notation**. Computers “prefer” **postfix notation**, in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as 3 4 + and 7 9 /, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation and evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, and in each algorithm, the stack is used for a different purpose.

In this exercise, you will write a Java version of the infix-to-postfix conversion algorithm. In the next exercise, you will write a Java version of the postfix expression evaluation algorithm. In a later exercise, you will discover that code you write in this exercise can help you implement a complete working compiler.

Write class `InfixToPostfixConverter` to convert an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

\[(6 + 2) * 5 - 8 / 4\]

to a postfix expression. The postfix version of the preceding infix expression is (note that no parenthesis are needed)

\[6 2 + 5 * 8 4 / -\]

The program should read the expression into `StringBuffer infix` and use one of the stack classes implemented in this chapter to help create the postfix expression in `StringBuffer postfix`. The algorithm for creating a postfix expression is as follows:

a) Push a left parenthesis '(' on the stack.

b) Append a right parenthesis ')' to the end of `infix`.

c) While the stack is not empty, read `infix` from left to right and do the following:

   If the current character in `infix` is a digit, append it to `postfix`.
   If the current character in `infix` is a left parenthesis, push it onto the stack.
   If the current character in `infix` is an operator:
      Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to `postfix`.
      Push the current character in `infix` onto the stack.
If the current character in **infix** is a right parenthesis:

Pop operators from the top of the stack and append them to **postfix** until a left parenthesis is at the top of the stack.

Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

+ addition
- subtraction
* multiplication
/ division
^ exponentiation
% modulus

The stack should be maintained with stack nodes that each contain an instance variable and a reference to the next stack node. Some of the methods you may want to provide are as follows:

a) Method **convertToPostfix**, which converts the infix expression to postfix notation.

b) Method **isOperator**, which determines whether c is an operator.

c) Method **precedence**, which determines if the precedence of **operator1** (from the infix expression) is less than, equal to or greater than the precedence of **operator2** (from the stack). The method returns **true** if **operator1** has lower precedence than **operator2**. Otherwise, **false** is returned.

d) Method **stackTop** (this should be added to the stack class), which returns the top value of the stack without popping the stack.

19.13 Write class **PostfixEvaluator**, which evaluates a postfix expression (assume it is valid) such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a **String-Buffer**. Using modified versions of the stack methods implemented earlier in this chapter, the program should scan the expression and evaluate it. The algorithm is as follows:

a) Append a right parenthesis (')') to the end of the postfix expression. When the right-parenthesis character is encountered, no further processing is necessary.

b) When the right-parenthesis character has not been encountered, read the expression from left to right.

If the current character is a digit do the following:

Push its integer value on the stack (the integer value of a digit character is its value in the computer's character set minus the value of '0' in Unicode).

Otherwise, if the current character is an **operator**:

Pop the two top elements of the stack into variables x and y.

Calculate y operator x.

Push the result of the calculation onto the stack.

c) When the right parenthesis is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[Note: In b) above (based on the sample expression at the beginning of this exercises), if the operator is '/ ', the top of the stack is 2 and the next element in the stack is 8, then pop 2 into x, pop 8 into y, evaluate 8 / 2 and push the result, 4, back on the stack. This note also applies to operator '-'.]

The arithmetic operations allowed in an expression are:

+ addition
- subtraction
* multiplication
/ division
The stack should be maintained with one of the stack classes introduced in this chapter. You may want to provide the following methods:

a) Method `evaluatePostfixExpression`, which evaluates the postfix expression.
b) Method `calculate`, which evaluates the expression `op1 operator op2`.
c) Method `push`, which pushes a value onto the stack.
d) Method `pop`, which pops a value off the stack.
e) Method `isEmpty`, which determines whether the stack is empty.
f) Method `printStack`, which prints the stack.

19.14 Modify the postfix evaluator program of Exercise 19.13 so that it can process integer operands larger than 9.

19.15 (Supermarket Simulation) Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of from 1 to 4 minutes. Also, each customer is serviced in random integer intervals of from 1 to 4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with “balanced” rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes), using the following algorithm:

a) Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.
b) At the first customer’s arrival time, do the following:
   - Determine customer’s service time (random integer from 1 to 4).
   - Begin servicing the customer.
   - Schedule the arrival time of the next customer (random integer 1 to 4 added to the current time).
c) For each minute of the day, consider the following:
   - If the next customer arrives, proceed as follows:
     - Say so.
     - Enqueue the customer.
     - Schedule the arrival time of the next customer.
   - If service was completed for the last customer, do the following:
     - Say so.
     - Dequeue next customer to be serviced.
     - Determine customer’s service completion time (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:

a) What is the maximum number of customers in the queue at any time?
b) What is the longest wait any one customer experiences?
c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?

19.16 Modify the program of Fig. 19.17 and Fig. 19.18 to allow the binary tree to contain duplicates.

19.17 Write a program based on the program of Fig. 19.17 and Fig. 19.18 that inputs a line of text, tokenizes the sentence into separate words (you might want to use the `StreamTokenizer` class from the `java.io` package), inserts the words in a binary search tree and prints the inorder, preorder and post-order traversals of the tree.

19.18 In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you would perform duplicate elimination when using only a single-subscripted array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.
19.19 Write a method `depth` that receives a binary tree and determines how many levels it has.

19.20 *(Recursively Print a List Backwards)* Write a method `printListBackwards` that recursively outputs the items in a linked list object in reverse order. Write a test program that creates a sorted list of integers and prints the list in reverse order.

19.21 *(Recursively Search a List)* Write a method `searchList` that recursively searches a linked list object for a specified value. Method `searchList` should return a reference to the value if it is found; otherwise, null should be returned. Use your method in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

19.22 *(Binary Tree Delete)* In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. There are three cases that are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child or the item is contained in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the reference in the parent node is set to null.

If the item to be deleted is contained in a node with one child, the reference in the parent node is set to reference the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the reference in the parent node cannot simply be assigned to reference one of the children of the node to be deleted. In most cases, the resulting binary search tree would not adhere to the following characteristic of binary search trees (with no duplicate values): *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.*

Which node is used as a *replacement node* to maintain this characteristic? It is either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let us consider the node with the smaller value. In a binary search tree, the largest value less than a parent’s value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the reference to the right child of the current node is null. We are now referencing the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node to null.
d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node’s position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node reference to the left child of the replacement node.
d) Set the reference to the right subtree in the replacement node reference to the right subtree of the node to be deleted.

e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

Write method `deleteNode`, which takes as its argument the value to be deleted. Method `deleteNode` should locate in the tree the node containing the value to be deleted and use the algorithms discussed here to delete the node. If the value is not found in the tree, the method should print a message that indicates whether the value is deleted. Modify the program of Fig. 19.17 and Fig. 19.18 to use this method. After deleting an item, call the methods `inorderTraversal`, `preorderTraversal` and `postorderTraversal` to confirm that the delete operation was performed correctly.

19.23 (Binary Tree Search) Write method `binaryTreeSearch`, which attempts to locate a specified value in a binary search tree object. The method should take as an argument a search key to be located. If the node containing the search key is found, the method should return a reference to that node; otherwise, the method should return a null reference.

19.24 (Level-Order Binary Tree Traversal) The program of Fig. 19.17 and Fig. 19.18 illustrated three recursive methods of traversing a binary tree—inorder, preorder and postorder traversals. This exercise presents the level-order traversal of a binary tree, in which the node values are printed level-by-level, starting at the root node level. The nodes on each level are printed from left to right. The level-order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

a) Insert the root node in the queue.

b) While there are nodes left in the queue, do the following:
   Get the next node in the queue.
   Print the node’s value.
   If the reference to the left child of the node is not null:
   Insert the left child node in the queue.
   If the reference to the right child of the node is not null:
   Insert the right child node in the queue.

Write method `levelOrder` to perform a level-order traversal of a binary tree object. Modify the program of Fig. 19.17 and Fig. 19.18 to use this method. [Note: You will also need to use queue-processing methods of Fig. 19.13 in this program.]

19.25 (Printing Trees) Write a recursive method `outputTree` to display a binary tree object on the screen. The method should output the tree row-by-row, with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree illustrated in Fig. 19.20 is output as shown in Fig. 19.21.

Note that the rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column of output starts five spaces to the right of the preceding column. Method `outputTree` should receive an argument `totalSpaces` representing the number of spaces preceding the value to be output. (This variable should start at zero so the root node is output at the left of the screen.) The method uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

While the reference to the current node is not null, perform the following:
Recursively call `outputTree` with the right subtree of the current node and `totalSpaces + 5`.
Use a for structure to count from 1 to `totalSpaces` and output spaces.
Output the value in the current node.
Set the reference to the current node to refer to the left subtree of the current node.
Increment `totalSpaces` by 5.
SPECIAL SECTION: BUILDING YOUR OWN COMPILER

In Exercise 7.42 and Exercise 7.43, we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute programs written in SML. In this section, we build a compiler that converts programs written in a high-level programming language to SML. This section “ties” together the entire programming process. You will write programs in this new high-level language, compile these programs on the compiler you build and run the programs on the simulator you built in Exercise 7.43. You should make every effort to implement your compiler in an object-oriented manner.

19.26 (The Simple Language) Before we begin building the compiler, we discuss a simple, yet powerful high-level language similar to early versions of the popular language Basic. We call the language Simple. Every Simple statement consists of a line number and a Simple instruction. Line numbers must appear in ascending order. Each instruction begins with one of the following Simple commands: rem, input, let, print, goto, if/goto or end (see Fig. 19.22). All commands except end can be used repeatedly. Simple evaluates only integer expressions using the +, -, *, and / operators. These operators have the same precedence as in Java. Parentheses can be used to change the order of evaluation of an expression.

<table>
<thead>
<tr>
<th>Command</th>
<th>Example statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rem</td>
<td>50 rem this is a remark</td>
<td>Any text following the command rem is for documentation purposes only and is ignored by the compiler.</td>
</tr>
<tr>
<td>input</td>
<td>30 input x</td>
<td>Display a question mark to prompt the user to enter an integer. Read that integer from the keyboard and store the integer in x.</td>
</tr>
<tr>
<td>let</td>
<td>80 let u = 4 * (j - 56)</td>
<td>Assign u the value of 4 * (j - 56). Note that an arbitrarily complex expression can appear to the right of the equal sign.</td>
</tr>
</tbody>
</table>

Fig. 19.21 Sample output of recursive method outputTree.

Fig. 19.22 Simple commands (part 1 of 2).
Our Simple compiler recognizes only lowercase letters. All characters in a Simple file should be lowercase. (Uppercase letters result in a syntax error unless they appear in a rem statement, in which case they are ignored.) A variable name is a single letter. Simple does not allow descriptive variable names, so variables should be explained in remarks to indicate their use in a program. Simple uses only integer variables. Simple does not have variable declarations—merely mentioning a variable name in a program causes the variable to be declared and initialized to zero. The syntax of Simple does not allow string manipulation (reading a string, writing a string, comparing strings etc.). If a string is encountered in a Simple program (after a command other than rem), the compiler generates a syntax error. The first version of our compiler assumes that Simple programs are entered correctly. Exercise 19.29 asks the reader to modify the compiler to perform syntax error checking.

Simple uses the conditional if/goto and unconditional goto statements to alter the flow of control during program execution. If the condition in the if/goto statement is true, control is transferred to a specific line of the program. The following relational and equality operators are valid in an if/goto statement: <, >, <=, >=, == or !=. The precedence of these operators is the same as in Java.

Let us now consider several programs that demonstrate Simple’s features. The first program (Fig. 19.23) reads two integers from the keyboard, stores the values in variables a and b and computes and prints their sum (stored in variable c).

The program of Fig. 19.24 determines and prints the larger of two integers. The integers are input from the keyboard and stored in s and t. The if/goto statement tests the condition s >= t. If the condition is true, control is transferred to line 90 and s is output; otherwise, t is output and control is transferred to the end statement in line 99, where the program terminates.

<table>
<thead>
<tr>
<th>Command</th>
<th>Example statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>10 print w</td>
<td>Display the value of w.</td>
</tr>
<tr>
<td>goto</td>
<td>70 goto 45</td>
<td>Transfer program control to line 45.</td>
</tr>
<tr>
<td>if/goto</td>
<td>35 if i == z goto 80</td>
<td>Compare i and z for equality and transfer program control to line 80 if the condition is true; otherwise, continue execution with the next statement.</td>
</tr>
<tr>
<td>end</td>
<td>99 end</td>
<td>Terminate program execution.</td>
</tr>
</tbody>
</table>

Fig. 19.22  Simple commands (part 2 of 2).

Our Simple compiler recognizes only lowercase letters. All characters in a Simple file should be lowercase. (Uppercase letters result in a syntax error unless they appear in a rem statement, in which case they are ignored.) A variable name is a single letter. Simple does not allow descriptive variable names, so variables should be explained in remarks to indicate their use in a program. Simple uses only integer variables. Simple does not have variable declarations—merely mentioning a variable name in a program causes the variable to be declared and initialized to zero. The syntax of Simple does not allow string manipulation (reading a string, writing a string, comparing strings etc.). If a string is encountered in a Simple program (after a command other than rem), the compiler generates a syntax error. The first version of our compiler assumes that Simple programs are entered correctly. Exercise 19.29 asks the reader to modify the compiler to perform syntax error checking.

Simple uses the conditional if/goto and unconditional goto statements to alter the flow of control during program execution. If the condition in the if/goto statement is true, control is transferred to a specific line of the program. The following relational and equality operators are valid in an if/goto statement: <, >, <=, >=, == or !=. The precedence of these operators is the same as in Java.

Let us now consider several programs that demonstrate Simple’s features. The first program (Fig. 19.23) reads two integers from the keyboard, stores the values in variables a and b and computes and prints their sum (stored in variable c).

The program of Fig. 19.24 determines and prints the larger of two integers. The integers are input from the keyboard and stored in s and t. The if/goto statement tests the condition s >= t. If the condition is true, control is transferred to line 90 and s is output; otherwise, t is output and control is transferred to the end statement in line 99, where the program terminates.

Fig. 19.23  Simple program that determines the sum of two integers.
Simple does not provide a repetition structure (such as Java’s for, while or do/while). However, Simple can simulate each of Java’s repetition structures by using the if/goto and goto statements. Figure 19.25 uses a sentinel-controlled loop to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable j. If the value entered is the sentinel value -9999, control is transferred to line 99, where the program terminates. Otherwise, k is assigned the square of j, k is output to the screen and control is passed to line 20, where the next integer is input.

Using the sample programs of Fig. 19.23–Fig. 19.25 as your guide, write a Simple program to accomplish each of the following:

a) Input three integers, determine their average and print the result.
b) Use a sentinel-controlled loop to input 10 integers and compute and print their sum.
c) Use a counter-controlled loop to input 7 integers, some positive and some negative, and compute and print their average.
d) Input a series of integers and determine and print the largest. The first integer indicates how many numbers should be processed.
e) Input 10 integers and print the smallest.
f) Calculate and print the sum of the even integers from 2 to 30.
g) Calculate and print the product of the odd integers from 1 to 9.

<table>
<thead>
<tr>
<th>1</th>
<th>10 rem   determine and print the larger of two integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20 input s</td>
</tr>
<tr>
<td>3</td>
<td>30 input t</td>
</tr>
<tr>
<td>4</td>
<td>32 rem</td>
</tr>
<tr>
<td>5</td>
<td>35 rem   test if s &gt;= t</td>
</tr>
<tr>
<td>6</td>
<td>40 if s &gt;= t goto 90</td>
</tr>
<tr>
<td>7</td>
<td>45 rem</td>
</tr>
<tr>
<td>8</td>
<td>50 rem   t is greater than s, so print t</td>
</tr>
<tr>
<td>9</td>
<td>60 print t</td>
</tr>
<tr>
<td>10</td>
<td>70 goto 99</td>
</tr>
<tr>
<td>11</td>
<td>75 rem</td>
</tr>
<tr>
<td>12</td>
<td>80 rem   s is greater than or equal to t, so print s</td>
</tr>
<tr>
<td>13</td>
<td>90 print s</td>
</tr>
<tr>
<td>14</td>
<td>99 end</td>
</tr>
</tbody>
</table>

Fig. 19.24 Simple program that finds the larger of two integers.

<table>
<thead>
<tr>
<th>1</th>
<th>10 rem   determine and print the larger of two integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20 input j</td>
</tr>
<tr>
<td>3</td>
<td>23 rem</td>
</tr>
<tr>
<td>4</td>
<td>25 rem   test for sentinel value</td>
</tr>
<tr>
<td>5</td>
<td>30 if j == -9999 goto 99</td>
</tr>
<tr>
<td>6</td>
<td>33 rem</td>
</tr>
<tr>
<td>7</td>
<td>35 rem   calculate square of j and assign result to k</td>
</tr>
<tr>
<td>8</td>
<td>40 let k = j * j</td>
</tr>
<tr>
<td>9</td>
<td>50 print k</td>
</tr>
<tr>
<td>10</td>
<td>53 rem</td>
</tr>
<tr>
<td>11</td>
<td>55 rem   loop to get next j</td>
</tr>
<tr>
<td>12</td>
<td>60 goto 20</td>
</tr>
<tr>
<td>13</td>
<td>99 end</td>
</tr>
</tbody>
</table>

Fig. 19.25 Calculate the squares of several integers.
19.27  (Building A Compiler; Prerequisite: Complete Exercise 7.42, Exercise 7.43, Exercise 19.12, Exercise 19.13 and Exercise 19.26) Now that the Simple language has been presented (Exercise 19.26), we discuss how to build a Simple compiler. First, we consider the process by which a Simple program is converted to SML and executed by the Simpletron simulator (see Fig. 19.26). A file containing a Simple program is read by the compiler and converted to SML code. The SML code is output to a file on disk, in which SML instructions appear one per line. The SML file is then loaded into the Simpletron simulator, and the results are sent to a file on disk and to the screen. Note that the Simpletron program developed in Exercise 7.43 took its input from the keyboard. It must be modified to read from a file so it can run the programs produced by our compiler.

The Simple compiler performs two passes of the Simple program to convert it to SML. The first pass constructs a symbol table in which every line number, variable name and constant of the Simple program is stored with its type and corresponding location in the final SML code (the symbol table is discussed in detail below). The first pass also produces the corresponding SML instruction object(s) for each of the Simple statements. If the Simple program contains statements that transfer control to a line later in the program, the first pass results in an SML program containing some “unfinished” instructions. The second pass of the compiler locates and completes the unfinished instructions and outputs the SML program to a file.

First Pass
The compiler begins by reading one statement of the Simple program into memory. The line must be separated into its individual tokens (i.e., “pieces” of a statement) for processing and compilation. (The StreamTokenizer class from the java.io package can be used.) Recall that every statement begins with a line number followed by a command. As the compiler breaks a statement into tokens, if the token is a line number, a variable or a constant, it is placed in the symbol table. A line number is placed in the symbol table only if it is the first token in a statement. The symbolTable object is an array of tableEntry objects representing each symbol in the program. There is no restriction on the number of symbols that can appear in the program. Therefore, the symbolTable for a particular program could be large. Make the symbolTable a 100-element array for now. You can increase or decrease its size once the program is working.

Each tableEntry object contains three members. Member symbol is an integer containing the Unicode representation of a variable (remember that variable names are single characters), a line number or a constant. Member type is one of the following characters indicating the symbol’s type: ‘C’ for constant, ‘L’ for line number or ‘V’ for variable. Member location contains the Simpletron memory location (00 to 99) to which the symbol refers. Simpletron memory is an array of 100 integers in which SML instructions and data are stored. For a line number, the location is the element in the Simpletron memory array at which the SML instructions for the Simple statement begin. For a variable or constant, the location is the element in the Simpletron memory array in which the variable or constant is stored. Variables and constants are allocated from the end of Simpletron’s memory backwards. The first variable or constant is stored at location 99, the next at location 98, etc.

The symbol table plays an integral part in converting Simple programs to SML. We learned in Chapter 7 that an SML instruction is a four-digit integer comprised of two parts—the operation code and the operand. The operation code is determined by commands in Simple. For example, the simple command input corresponds to SML operation code 10 (read), and the Simple command print corresponds to SML operation code 11 (write). The operand is a memory location containing the data on which the operation code performs its task (e.g., operation code 10 reads a value from the keyboard and stores it in the memory location specified by the operand). The compiler searches symbolTable to determine the Simpletron memory location for each symbol, so the corresponding location can be used to complete the SML instructions.
The compilation of each Simple statement is based on its command. For example, after the line number in a *rem* statement is inserted in the symbol table, the remainder of the statement is ignored by the compiler because a remark is for documentation purposes only. The *input, print, goto* and *end* statements correspond to the SML *read, write, branch* (to a specific location) and *halt* instructions. Statements containing these Simple commands are converted directly to SML. (*Note: A goto statement may contain an unresolved reference if the specified line number refers to a statement further into the Simple program file; this is sometimes called a forward reference.*)

When a *goto* statement is compiled with an unresolved reference, the SML instruction must be flagged to indicate that the second pass of the compiler must complete the instruction. The flags are stored in a 100-element array *flags* of type *int* in which each element is initialized to *-1*. If the memory location to which a line number in the Simple program refers is not yet known (i.e., it is not in the symbol table), the line number is stored in array *flags* in the element with the same subscript as the incomplete instruction. The operand of the incomplete instruction is set to *00* temporarily. For example, an unconditional branch instruction (making a forward reference) is left as *+4000* until the second pass of the compiler. The second pass of the compiler will be described shortly.

Compilation of *if/goto* and *let* statements is more complicated than other statements—they are the only statements that produce more than one SML instruction. For an *if/goto* statement, the compiler produces code to test the condition and to branch to another line if necessary. The result of the branch could be an unresolved reference. Each of the relational and equality operators can be simulated by using SML’s *branch zero* and *branch negative* instructions (or possibly a combination of both).

For a *let* statement, the compiler produces code to evaluate an arbitrarily complex arithmetic expression consisting of integer variables and/or constants. Expressions should separate each operand and operator with spaces. Exercise 19.12 and Exercise 19.13 presented the infix-to-postfix conversion algorithm and the postfix evaluation algorithm used by compilers to evaluate expressions. Before proceeding with your compiler, you should complete each of these exercises. When a compiler encounters an expression, it converts the expression from infix notation to postfix notation, then evaluates the postfix expression.

How is it that the compiler produces the machine language to evaluate an expression containing variables? The postfix evaluation algorithm contains a “hook” where the compiler can generate SML instructions rather than actually evaluating the expression. To enable this “hook” in the compiler, the postfix evaluation algorithm must be modified to search the symbol table for each symbol it encounters (and possibly insert it), determine the symbol’s corresponding memory location and push the memory location on the stack (instead of the symbol). When an operator is encountered in the postfix expression, the two memory locations at the top of the stack are popped, and machine language for effecting the operation is produced by using the memory locations as operands. The result of each subexpression is stored in a temporary location in memory and pushed back onto the stack so the
evaluation of the postfix expression can continue. When postfix evaluation is complete, the memory location containing the result is the only location left on the stack. This is popped, and SML instructions are generated to assign the result to the variable at the left of the \texttt{let} statement.

\textbf{Second Pass}

The second pass of the compiler performs two tasks: Resolve any unresolved references and output the SML code to a file. Resolution of references occurs as follows:

\begin{itemize}
  \item[a)] Search the \texttt{flags} array for an unresolved reference (i.e., an element with a value other than -1).
  \item[b)] Locate the object in array \texttt{symbolTable} containing the symbol stored in the \texttt{flags} array (be sure that the type of the symbol is 'L' for line number).
  \item[c)] Insert the memory location from member \texttt{location} into the instruction with the unresolved reference (remember that an instruction containing an unresolved reference has operand 00).
  \item[d)] Repeat steps (a), (b) and (c) until the end of the \texttt{flags} array is reached.
\end{itemize}

After the resolution process is complete, the entire array containing the SML code is output to a disk file with one SML instruction per line. This file can be read by the Simpletron for execution (after the simulator is modified to read its input from a file). Compiling your first Simple program into an SML file and executing that file should give you a real sense of personal accomplishment.

\textbf{A Complete Example}

The following example illustrates complete conversion of a Simple program to SML as it will be performed by the Simple compiler. Consider a Simple program that inputs an integer and sums the values from 1 to that integer. The program and the SML instructions produced by the first pass of the Simple compiler are illustrated in Fig. 19.27. The symbol table constructed by the first pass is shown in Fig. 19.28.

\begin{tabular}{|l|c|c|}
\hline
Simple program & SML location and instruction & Description \\
\hline
5 \texttt{rem sum 1 to x} & \texttt{none} & \texttt{rem} ignored \\
10 \texttt{input x} & 00 +1099 & read \texttt{x} into location 99 \\
15 \texttt{rem check y == x} & \texttt{none} & \texttt{rem} ignored \\
20 \texttt{if y == x goto 60} & 01 +2098 & load \texttt{y(98)} into accumulator \\
 & 02 +3199 & sub \texttt{x(99)} from accumulator \\
 & 03 +4200 & branch zero to unresolved location \\
25 \texttt{rem increment y} & \texttt{none} & \texttt{rem} ignored \\
30 \texttt{let y = y + 1} & 04 +2098 & load \texttt{y} into accumulator \\
 & 05 +3097 & add \texttt{1(97)} to accumulator \\
 & 06 +2196 & store in temporary location 96 \\
 & 07 +2096 & load from temporary location 96 \\
 & 08 +2198 & store accumulator in \texttt{y} \\
35 \texttt{rem add y to total} & \texttt{none} & \texttt{rem} ignored \\
\hline
\end{tabular}

\textit{Fig. 19.27} SML instructions produced after the compiler’s first pass (part 1 of 2).
Most Simple statements convert directly to single SML instructions. The exceptions in this program are remarks, the *if/goto* statement in line 20 and the *let* statements. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in
case the line number is referenced in a `goto` statement or an `if/goto` statement. Line 20 of the program specifies that, if the condition `y == x` is true, program control is transferred to line 60. Since line 60 appears later in the program, the first pass of the compiler has not as yet placed 60 in the symbol table. (Statement line numbers are placed in the symbol table only when they appear as the first token in a statement.) Therefore, it is not possible at this time to determine the operand of the SML `branch zero` instruction at location 03 in the array of SML instructions. The compiler places 60 in location 03 of the `flags` array to indicate that the second pass completes this instruction.

We must keep track of the next instruction location in the SML array because there is not a one-to-one correspondence between Simple statements and SML instructions. For example, the `if/goto` statement of line 20 compiles into three SML instructions. Each time an instruction is produced, we must increment the `instruction counter` to the next location in the SML array. Note that the size of Simpletron’s memory could present a problem for Simple programs with many statements, variables and constants. It is conceivable that the compiler will run out of memory. To test for this case, your program should contain a `data counter` to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the value of the data counter, the SML array is full. In this case, the compilation process should terminate, and the compiler should print an error message indicating that it ran out of memory during compilation. This serves to emphasize that, although the programmer is freed from the burdens of managing memory by the compiler, the compiler itself must carefully determine the placement of instructions and data in memory and must check for such errors as memory being exhausted during the compilation process.

**A Step-by-Step View of the Compilation Process**

Let us now walk through the compilation process for the Simple program in Fig. 19.27. The compiler reads the first line of the program

```
5 rem sum 1 to x
```

into memory. The first token in the statement (the line number) is determined using the `StreamTokenizer` class (see Chapter 10 for a discussion of Java’s string manipulation methods). The token returned by the `StreamTokenizer` is converted to an integer by using `static` method `Integer.parseInt()`, so the symbol 5 can be located in the symbol table. If the symbol is not found, it is inserted into the symbol table.

We are at the beginning of the program and this is the first line, and no symbols are in the table yet. Therefore, 5 is inserted into the symbol table as type `L` (line number) and assigned the first location in SML array (00). Although this line is a remark, a space in the symbol table is still allocated for the line number (in case it is referenced by a `goto` or an `if/goto`). No SML instruction is generated for a `rem` statement, so the instruction counter is not incremented.

```
10 input x
```

is tokenized next. The line number 10 is placed in the symbol table as type `L` and assigned the first location in the SML array (00 because a remark began the program, so the instruction counter is currently 00). The command `input` indicates that the next token is a variable (only a variable can appear in an `input` statement). `input` corresponds directly to an SML operation code; therefore, the compiler simply has to determine the location of `x` in the SML array. Symbol `x` is not found in the symbol table. So, it is inserted into the symbol table as the Unicode representation of `x`, given type `V` and assigned location 99 in the SML array (data storage begins at 99 and is allocated backwards). SML code can now be generated for this statement. Operation code 10 (the SML read operation code) is multiplied by 100, and the location of `x` (as determined in the symbol table) is added to complete the instruction. The instruction is then stored in the SML array at location 00. The instruction counter is incremented by one, because a single SML instruction was produced.
The statement

\[ 15 \text{ rem check } y == x \]

is tokenized next. The symbol table is searched for line number 15 (which is not found). The line number is inserted as type L and assigned the next location in the array, 01. (Remember that rem statements do not produce code, so the instruction counter is not incremented.)

The statement

\[ 20 \text{ if } y == x \text{ goto 60} \]

is tokenized next. Line number 20 is inserted in the symbol table and given type L at the next location in the SML array 01. The command if indicates that a condition is to be evaluated. The variable y is not found in the symbol table, so it is inserted and given the type V and the SML location 98. Next, SML instructions are generated to evaluate the condition. There is no direct equivalent in SML for the if/goto; it must be simulated by performing a calculation using x and y and branching according to the result. If y is equal to x, the result of subtracting x from y is zero, so the branch zero instruction can be used with the result of the calculation to simulate the if/goto statement. The first step requires that y be loaded (from SML location 98) into the accumulator. This produces the instruction 01 +2098. Next, x is subtracted from the accumulator. This produces the instruction 02 +3199. The value in the accumulator may be zero, positive or negative. The operator is ==, so we want to branch zero. First, the symbol table is searched for the branch location (60 in this case), which is not found. So, 60 is placed in the flags array at location 03, and the instruction 03 +4200 is generated. (We cannot add the branch location because we have not yet assigned a location to line 60 in the SML array.) The instruction counter is incremented to 04.

The compiler proceeds to the statement

\[ 25 \text{ rem increment } y \]

The line number 25 is inserted in the symbol table as type L and assigned SML location 04. The instruction counter is not incremented.

When the statement

\[ 30 \text{ let } y = y + 1 \]

is tokenized, the line number 30 is inserted in the symbol table as type L and assigned SML location 04. Command let indicates that the line is an assignment statement. First, all the symbols on the line are inserted in the symbol table (if they are not already there). The integer 1 is added to the symbol table as type C and assigned SML location 97. Next, the right side of the assignment is converted from infix to postfix notation. Then the postfix expression \((y 1 +)\) is evaluated. Symbol y is located in the symbol table, and its corresponding memory location is pushed onto the stack. Symbol 1 is also located in the symbol table, and its corresponding memory location is pushed onto the stack. When the operator + is encountered, the postfix evaluator pops the stack into the right operand of the operator and pops the stack again into the left operand of the operator, then produces the SML instructions

\[
\begin{align*}
04 & +2098 \quad \text{(load y)} \\
05 & +3097 \quad \text{(add 1)}
\end{align*}
\]

The result of the expression is stored in a temporary location in memory (96) with instruction

\[ 06 +2196 \quad \text{(store temporary)} \]

and the temporary location is pushed onto the stack. Now that the expression has been evaluated, the result must be stored in \(y\) (i.e., the variable on the left side of \(=\)). So, the temporary location is loaded into the accumulator and the accumulator is stored in \(y\) with the instructions
The reader will immediately notice that SML instructions appear to be redundant. We will discuss this issue shortly.

When the statement

```plaintext
35 rem   add y to total
```

is tokenized, line number 35 is inserted in the symbol table as type $L$ and assigned location 09.

The statement

```plaintext
40 let t = t + y
```

is similar to line 30. The variable $t$ is inserted in the symbol table as type $V$ and assigned SML location 95. The instructions follow the same logic and format as line 30, and the instructions 09 $+2095, 10 +3098, 11 +2194, 12 +2094$ and 13 $+2195$ are generated. Note that the result of $t + y$ is assigned to temporary location 94 before being assigned to $t$ (95). Once again, the reader will note that the instructions in memory locations 11 and 12 appear to be redundant. Again, we will discuss this shortly.

The statement

```plaintext
45 rem   loop y
```

is a remark, so line 45 is added to the symbol table as type $L$ and assigned SML location 14.

The statement

```plaintext
50 goto 20
```

transfers control to line 20. Line number 50 is inserted in the symbol table as type $L$ and assigned SML location 14. The equivalent of `goto` in SML is the *unconditional branch* (40) instruction that transfers control to a specific SML location. The compiler searches the symbol table for line 20 and finds that it corresponds to SML location 01. The operation code (40) is multiplied by 100, and location 01 is added to it to produce the instruction 14 $+4001$.

The statement

```plaintext
55 rem   output result
```

is a remark, so line 55 is inserted in the symbol table as type $L$ and assigned SML location 15.

The statement

```plaintext
60 print t
```

is an output statement. Line number 60 is inserted in the symbol table as type $L$ and assigned SML location 15. The equivalent of `print` in SML is operation code 11 *(write)*. The location of $t$ is determined from the symbol table and added to the result of the operation code multiplied by 100.

The statement

```plaintext
99 end
```

is the final line of the program. Line number 99 is stored in the symbol table as type $L$ and assigned SML location 16. The `end` command produces the SML instruction $+4300$ (43 is *halt* in SML), which is written as the final instruction in the SML memory array.

This completes the first pass of the compiler. We now consider the second pass. The *flags* array is searched for values other than $-1$. Location 03 contains 60, so the compiler knows that instruction 03 is incomplete. The compiler completes the instruction by searching the symbol
table for 60, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line 60 corresponds to SML location 15, so the completed instruction 03 +4215 is produced, replacing 03 +4200. The Simple program has now been compiled successfully.

To build the compiler, you will have to perform each of the following tasks:

a) Modify the Simpletron simulator program you wrote in Exercise 7.43 to take its input from a file specified by the user (see Chapter 16). The simulator should output its results to a disk file in the same format as the screen output. Convert the simulator to be an object-oriented program. In particular, make each part of the hardware an object. Arrange the instruction types into a class hierarchy using inheritance. Then execute the program polymorphically simply by telling each instruction to execute itself with an executeInstruction message.

b) Modify the infix-to-postfix evaluation algorithm of Exercise 19.12 to process multidigit integer operands and single-letter variable name operands. (Hint: Class StreamTokenizer can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers by using Integer class method parseInt.) [Note: The data representation of the postfix expression must be altered to support variable names and integer constants.]

c) Modify the postfix evaluation algorithm to process multidigit integer operands and variable name operands. Also, the algorithm should now implement the “hook” discussed earlier so that SML instructions are produced rather than directly evaluating the expression. (Hint: Class StreamTokenizer can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers by using Integer class method parseInt.) [Note: The data representation of the postfix expression must be altered to support variable names and integer constants.]

d) Build the compiler. Incorporate parts b) and c) for evaluating expressions in let statements. Your program should contain a method that performs the first pass of the compiler and a method that performs the second pass of the compiler. Both methods can call other methods to accomplish their tasks. Make your compiler as object oriented as possible.

19.28 (Optimizing the Simple Compiler) When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, usually in triplets called productions. A production normally consists of three instructions, such as load, add and store. For example, Fig. 19.29 illustrates five of the SML instructions that were produced in the compilation of the program in Fig. 19.27. The first three instructions are the production that adds 1 to y. Note that instructions 06 and 07 store the accumulator value in temporary location 96, then load the value back into the accumulator so instruction 08 can store the value in location 98. Often a production is followed by a load instruction for the same location that was just stored. This code can be optimized by eliminating the store instruction and the subsequent load instruction that operate on the same memory location, thus enabling the Simpletron to execute the program faster. Figure 19.30 illustrates the optimized SML for the program of Fig. 19.27. Note that there are four fewer instructions in the optimized code—a memory-space savings of 25%.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>04</td>
<td>+2098 (load)</td>
</tr>
<tr>
<td>2</td>
<td>05</td>
<td>+3097 (add)</td>
</tr>
<tr>
<td>3</td>
<td>06</td>
<td>+2196 (store)</td>
</tr>
<tr>
<td>4</td>
<td>07</td>
<td>+2096 (load)</td>
</tr>
<tr>
<td>5</td>
<td>08</td>
<td>+2198 (store)</td>
</tr>
</tbody>
</table>

Fig. 19.29 Unoptimized code from the program of Fig. 19.25.
Perform the following modifications to the Simple compiler. Some of these modifications might also require modifications to the Simpletron simulator program written in Exercise 7.43.

a) Allow the modulus operator (%) to be used in let statements. Simpletron Machine Language must be modified to include a modulus instruction.

b) Allow exponentiation in a let statement using ^ as the exponentiation operator. Simpletron Machine Language must be modified to include an exponentiation instruction.

c) Allow the compiler to recognize uppercase and lowercase letters in Simple statements (e.g., 'A' is equivalent to 'a'). No modifications to the Simpletron simulator are required.

d) Allow input statements to read values for multiple variables such as input x, y. No modifications to the Simpletron simulator are required to perform this enhancement to the Simple compiler.

e) Allow the compiler to output multiple values from a single print statement, such as print a, b, c. No modifications to the Simpletron simulator are required to perform this enhancement.

f) Add syntax-checking capabilities to the compiler so error messages are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron simulator are required.
g) Allow arrays of integers. No modifications to the Simpletron simulator are required to perform this enhancement.

h) Allow subroutines specified by the Simple commands `gosub` and `return`. Command `gosub` passes program control to a subroutine and command `return` passes control back to the statement after the `gosub`. This is similar to a method call in Java. The same subroutine can be called from many `gosub` commands distributed throughout a program. No modifications to the Simpletron simulator are required.

i) Allow repetition structures of the form

```
for x = 2 to 10 step 2
  Simple statements
next
```

This `for` statement loops from 2 to 10 with an increment of 2. The `next` line marks the end of the body of the `for` line. No modifications to the Simpletron simulator are required.

j) Allow repetition structures of the form

```
for x = 2 to 10
  Simple statements
next
```

This `for` statement loops from 2 to 10 with a default increment of 1. No modifications to the Simpletron simulator are required.

k) Allow the compiler to process string input and output. This requires the Simpletron simulator to be modified to process and store string values. [Hint: Each Simpletron word (i.e., memory location) can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the Unicode decimal equivalent of a character. Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the Simpletron word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one Unicode character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

l) Allow the compiler to process floating-point values in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.

19.30 (A Simple Interpreter) An interpreter is a program that reads a high-level language program statement, determines the operation to be performed by the statement and executes the operation immediately. The high-level language program is not converted into machine language first. Interpreters execute more slowly than compilers do, because each statement encountered in the program being interpreted must first be deciphered at execution time. If statements are contained in a loop, the statements are deciphered each time they are encountered in the loop. Early versions of the Basic programming language were implemented as interpreters. Most Java programs are run interpretively.

Write an interpreter for the Simple language discussed in Exercise 19.26. The program should use the infix-to-postfix converter developed in Exercise 19.12 and the postfix evaluator developed in Exercise 19.13 to evaluate expressions in a `let` statement. The same restrictions placed on the Simple language in Exercise 19.26 should be adhered to in this program. Test the interpreter with the Simple programs written in Exercise 19.26. Compare the results of running these programs in the interpreter with the results of compiling the Simple programs and running them in the Simpletron simulator built in Exercise 7.43.

19.31 (Insert/Delete Anywhere in a Linked List) Our linked-list class allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used inheritance or composition to produce a stack class and a queue class with a minimal amount
of code simply by reusing the list class. Linked lists are normally more general than those we provided. Modify the linked-list class we developed in this chapter to handle insertions and deletions anywhere in the list.

19.32 (Lists and Queues without Tail References) Our implementation of a linked list (Fig. 19.3) used both a `firstNode` and a `lastNode`. The `lastNode` was useful for the `insertAtBack` and `removeFromBack` methods of the `List` class. The `insertAtBack` method corresponds to the `enqueue` method of the `Queue` class.

Rewrite the `List` class so that it does not use a `lastNode`. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the `Queue` class (Fig. 19.13)?

19.33 (Performance of Binary Tree Sorting and Searching) One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree—for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data were inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?

19.34 (Indexed Lists) As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list-searching performance is to create and maintain an index to the list. An index is a set of references to key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries—one for each letter of the alphabet. A search operation for a last name beginning with ‘Y’ would then first search the index to determine where the ‘Y’ entries begin, then “jump into” the list at that point and search linearly until the desired name is found. This would be much faster than searching the linked list from the beginning. Use the `List` class of Fig. 19.3 as the basis of an `IndexedList` class.

Write a program that demonstrates the operation of indexed lists. Be sure to include methods `insertInIndexedList`, `searchIndexedList` and `deleteFromIndexedList`. 
Java Utilities Package and Bit Manipulation

Objectives

- To understand containers, such as classes Vector and Stack, and the Enumeration interface.
- To be able to create Hashtable objects and persistent hash tables called Properties objects.
- To understand random number generation with instances of class Random.
- To use bit manipulation and BitSet objects.

Nothing can have value without being an object of utility.
Karl Marx

I’ve been in Who’s Who, and I know what’s what, but this is the first time I ever made the dictionary.
Mae West

O! many a shaft at sent
Finds mark the archer little meant!
Sir Walter Scott

There was the Door to which I found no Key;
There was the Veil through which I might not see.
Edward FitzGerald, The Rubáiyát of Omar Khayyám, st. 32

“It’s a poor sort of memory that only works backwards,” the Queen remarked.
Lewis Carroll [Charles Lutwidge Dodgson]

Not by age but by capacity is wisdom acquired.
Titus Maccius Plautus, Trinummus, act II, sc. ii, 1.88
20.1 Introduction

This chapter discusses several utility classes and interfaces in package `java.util`, including class `Vector`, interface `Enumeration`, class `Stack`, class `Dictionary`, class `Hashtable`, class `Properties`, class `Random` and class `BitSet`.

Programs use class `Vector` to create array-like objects that can grow and shrink dynamically as a program’s data storage requirements change. We consider interface `Enumeration`, which enables a program to iterate through the elements of a container such as a `Vector`.

Class `Stack` offers conventional stack operations `push` and `pop`, as well as others we did not consider in Chapter 19.

Class `Dictionary` is an abstract class that provides a framework for storing keyed data in tables and retrieving that data. The chapter explains the theory of “hashing,” a technique for rapidly storing and retrieving information from tables, and demonstrates the construction and manipulation of hash tables with Java’s `Hashtable` class. Also, the chapter considers class `Properties`, which provides support for persistent hash tables—hash tables that can be written to a file with an output stream and read from a file with an input stream.

Class `Random` provides a richer collection of random number capabilities than is available with static method `random` of class `Math`.

The chapter presents an extensive discussion of bit manipulation operators, followed by a discussion of class `BitSet` that enables the creation of bit-array-like objects for setting and getting individual bit values.

Chapter 21, Collections, introduces a framework for manipulating groups of objects called `collections`. Objects of type `Vector`, `Stack` and `Hashtable` are collections.

20.2 Vector Class and Enumeration Interface

In most programming languages, including Java, conventional arrays are fixed in size—they cannot grow or shrink in response to an application’s changing storage requirements. Java class `Vector` provides the capabilities of array-like data structures that can resize themselves dynamically.
At any time, the Vector contains a certain number of elements less than or equal to its capacity. The capacity is the space that has been reserved for the Vector’s elements. If a Vector requires additional capacity, it grows by a capacity increment that you specify or by a default assumed by the system. If you do not specify a capacity increment, the system will double the size of the Vector each time additional capacity is needed.

**Performance Tip 20.1**

Inserting additional elements into a Vector whose current size is less than its capacity is a relatively fast operation.

**Performance Tip 20.2**

It is a relatively slow operation to insert an element into a Vector that needs to grow larger to accommodate the new element.

**Performance Tip 20.3**

The default capacity increment doubles the size of the Vector. This may seem a waste of storage, but it is actually an efficient way for many Vectors to grow quickly to be “about the right size.” This is much more efficient time-wise than growing the Vector each time by only as much space as it takes to hold a single element. This disadvantage is that the Vector might occupy more space than it requires.

**Performance Tip 20.4**

If storage is at a premium, use Vector method trimToSize to trim a Vector’s capacity to the Vector’s exact size. This optimizes a Vector’s use of storage. However, adding another element to the Vector will force the Vector to grow dynamically—trimming leaves no room for growth.

Vectors store references to Objects. Thus, a program can store references to any objects in a Vector. To store values of primitive data types in Vectors, use the type-wrapper classes (e.g., Integer, Long, Float) from package java.lang to create objects containing the primitive data type values.

Figure 20.1 demonstrates class Vector and several of its methods. The program provides a JButton that enables the user to test each of the methods. The user can type a String into the provided JTextField, and then press a button to see what the method does. Each operation displays a message in a JLabel to indicate the results of the operation.

```java
1 // Fig. 20.1: VectorTest.java
2 // Testing the Vector class of the java.util package
3
4 // Java core packages
5 import java.util.*;
6 import java.awt.*;
7 import java.awt.event.*;
8
9 // Java extension packages
10 import javax.swing.*;
11
12 public class VectorTest extends JFrame {
13     private JLabel statusLabel;
14 }
```

Fig. 20.1 Demonstrating class Vector of package java.util (part 1 of 6).
private Vector vector;
private JTextField inputField;

// set up GUI to test Vector methods
public VectorTest()
{
    super("Vector Example");
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );
    statusLabel = new JLabel();
    vector = new Vector(1);
    container.add( new JLabel( "Enter a string" ) );
    inputField = new JTextField( 10 );
    container.add( inputField );

    // button to add element to vector
    JButton addButton = new JButton( "Add" );
    addButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent event )
                {
                    // add an element to vector
                    vector.addElement( inputField.getText() );
                    statusLabel.setText( "Added to end: " +
                            inputField.getText() );
                    inputField.setText( "" );
                }
            }); // end call to addActionListener
    container.add( addButton );

    // button to remove element from vector
    JButton removeButton = new JButton( "Remove" );
    removeButton.addActionListener(
            new ActionListener() {
                public void actionPerformed( ActionEvent event )
                {
                    // remove element from vector
                    if ( vector.removeElement( inputField.getText() ) )
                        statusLabel.setText( "Removed: " +
                                inputField.getText() );
                }
            }); // end call to addActionListener
    container.add( removeButton );
}

Fig. 20.1 Demonstrating class Vector of package java.util (part 2 of 6).
else
    statusLabel.setText( inputField.getText() +
        " not in vector" );
}
}); // end call to addActionListener

container.add( removeButton );

// button to get first element of vector
JButton firstButton = new JButton( "First" );

firstButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            // return first element of vector
            try {
                statusLabel.setText( 
                    "First element: " + vector.firstElement() );
            }
            // catch exception if Vector empty
            catch ( NoSuchElementException exception ) {
                statusLabel.setText( exception.toString() );
            }
        }
    }); // end call to addActionListener

container.add( firstButton );

// button to get last element of vector
JButton lastButton = new JButton( "Last" );

lastButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            // return last element of vector
            try {
                statusLabel.setText( 
                    "Last element: " + vector.lastElement() );
            }
            // catch exception if Vector empty
            catch ( NoSuchElementException exception ) {
                statusLabel.setText( exception.toString() );
            }
        }
    }); // end call to addActionListener

container.add( lastButton );

Fig. 20.1 Demonstrating class Vector of package java.util (part 3 of 6).
// button to determine whether vector is empty
JButton emptyButton = new JButton( "Is Empty?" );

emptyButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // determine if Vector is empty
            statusLabel.setText( vector.isEmpty() ?
                "Vector is empty" : "Vector is not empty" );
        }
    }); // end call to addActionListener

card.add( emptyButton );

// button to determine whether vector contains search key
JButton containsButton = new JButton( "Contains" );

containsButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            String searchKey = inputField.getText();

            // determine if Vector contains searchKey
            if ( vector.contains( searchKey ) )
                statusLabel.setText( "Vector contains " + searchKey );
            else
                statusLabel.setText( "Vector does not contain " + searchKey );
        }
    }); // end call to addActionListener

card.add( containsButton );

// button to determine location of value in vector
JButton locationButton = new JButton( "Location" );

locationButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // determine if Vector contains searchKey
            if ( vector.contains( searchKey ) )
                statusLabel.setText( "Vector contains " + searchKey );
            else
                statusLabel.setText( "Vector does not contain " + searchKey );
        }
    }); // end call to addActionListener

card.add( locationButton );

Fig. 20.1  Demonstrating class Vector of package java.util (part 4 of 6).
new ActionListener() {

    public void actionPerformed( ActionEvent event )
    {
        // get location of an object in Vector
        statusLabel.setText( "Element is at location " +
                vector.indexOf( inputField.getText() ) );
    }
}; // end call to addActionListener

container.add( locationButton );

// button to trim vector size
JButton trimButton = new JButton( "Trim" );
trimButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            // remove unoccupied elements to save memory
            vector.trimToSize();
            statusLabel.setText( "Vector trimmed to size" );
        }
    });

container.add( trimButton );

// button to display vector size and capacity
JButton statsButton = new JButton( "Statistics" );
statsButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            // get size and capacity of Vector
            statusLabel.setText( "Size = " + vector.size() +
                    "; capacity = " + vector.capacity() );
        }
    }); // end call to addActionListener

container.add( statsButton );

// button to display vector contents
JButton displayButton = new JButton( "Display" );
displayButton.addActionListener(

Fig. 20.1 Demonstrating class Vector of package java.util (part 5 of 6).
new ActionListener() {
    public void actionPerformed( ActionEvent event )
    {
        // use Enumeration to output Vector contents
        Enumeration enum = vector.elements();
        StringBuffer buf = new StringBuffer();
        while ( enum.hasMoreElements() )
            buf.append( enum.nextElement() ).append( "  ");
        JOptionPane.showMessageDialog( null,
            buf.toString(), "Display",
            JOptionPane.PLAIN_MESSAGE );
    }
}; // end call to addActionListener

canvas.add( displayButton );
canvas.add( statusLabel );

setSize( 300, 200 );
setVisible( true );

} // end VectorTest constructor

// execute application
public static void main( String args[] )
{
    VectorTest application = new VectorTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

} // end class VectorTest

Fig. 20.1 Demonstrating class Vector of package java.util (part 6 of 6).

The application’s constructor creates a Vector (line 26) with an initial capacity of one element. This Vector will double in size each time it needs to grow to accommodate more elements. Class Vector provides three other constructors. The no-argument constructor creates an empty Vector with an initial capacity of 10 elements. The constructor that takes two
arguments creates a `Vector` with an *initial capacity* specified by the first argument and a *capacity increment* specified by the second argument. Each time the `Vector` needs to grow, it will add space for the specified number of elements in the capacity increment. The constructor that takes a `Collection` allows creates a copy of a collection’s elements and stores them in the `Vector`. In Chapter 21, we discuss `Collections`.

Line 43 calls `Vector` method `addElement` to add its argument to the end of the `Vector`. If necessary, the `Vector` increases its capacity to accommodate the new element. Class `Vector` also provides method `insertElementAt` to insert an element at a specified position in the `Vector` and method `setElementAt` to set the element at a specific position in the `Vector`. Method `insertElementAt` makes room for the new element by shifting elements. Method `setElementAt` replaces the element at the specified position with its argument.

Line 63 calls `Vector` method `removeElement` to remove the first occurrence of its argument from the `Vector`. The method returns `true` if it finds the element in the `Vector`; otherwise, the method returns `false`. If the element is removed, all elements after that element in the `Vector` shift one position toward the beginning of the `Vector` to fill in the position of the removed element. Class `Vector` also provides method `removeAllElements` to remove every element from the `Vector` and method `removeElementAt` to remove the element at a specified index.

Line 87 calls `Vector` method `firstElement` to return a reference to the first element in the `Vector`. This method throws a `NoSuchElementException` if there are no elements currently in the `Vector`. Line 112 calls `Vector` method `lastElement` to return a reference to the last element in the `Vector`. This method throws a `NoSuchElementException` if there are no elements currently in the `Vector`.

Line 135 calls `Vector` method `isEmpty` to determine whether the `Vector` is empty. The method returns `true` if there are no elements in the `Vector`; otherwise, the method returns `false`.

Line 155 calls `Vector` method `contains` to determine whether the `Vector` contains the `searchKey` specified as an argument. Method `contains` returns `true` if `searchKey` is in the `Vector`; otherwise, the method returns `false`. Method `contains` uses `Object` method `equals` to determine whether the `searchKey` is equal to one of the `Vector`’s elements. Many classes override method `equals` to perform the comparisons in a manner specific to those classes. For example, class `String` defines `equals` to compare the individual characters in the two `Strings` being compared. If method `equals` is not overridden, the original version from class `Object` is used. This version performs comparisons using operator `==` to determine whether two references refer to the same object in memory.

Line 178 calls `Vector` method `indexOf` to determine the index of the first location in the `Vector` containing the argument. The method returns `-1` if the argument is not found in the `Vector`. An overloaded version of this method takes a second argument specifying the index in the `Vector` at which the search should begin.

**Performance Tip 20.5**

`Vector` methods `contains` and `indexOf` perform linear searches of a `Vector`’s contents, which are inefficient for large `Vectors`. If a program frequently searches for elements in a collection, consider using a `Hashtable` (see Section 20.5) or one of the Java Collection API’s `Map` implementations (see Chapter 21).
Line 195 calls Vector method trimToSize to reduce the capacity of the Vector to the current number of elements in the Vector (i.e., the Vector’s current size).

Lines 213–214 use Vector methods size and capacity to determine the number of elements currently in the Vector and the number of elements that can be stored in the Vector without allocating more memory, respectively.

Line 231 calls Vector method elements to return an Enumeration that enables the program to iterate through the Vector’s elements. An Enumeration provides two methods—hasMoreElements and nextElement. In line 234, method hasMoreElements returns true if there are more elements in the Vector. In line 235, method nextElement returns a reference to the next element in the Vector. If there are no more elements, method nextElement throws a NoSuchElementException.

For complete information on class Vector and its other methods, see the online Java API documentation.

20.3 Stack Class

In Chapter 19, Data Structures, we learned how to build such fundamental data structures as linked lists, stacks, queues and trees. In a world of software reuse, instead of building data structures as we need them, often we can take advantage of existing data structures. In this section, we investigate class Stack in the Java utilities package (java.util).

In Section 20.2, we discussed class Vector, which implements a dynamically resiz-able array. Class Stack extends class Vector to implement a stack data structure. As does Vector, class Stack stores references to Objects. To store primitive data types, use the appropriate type-wrapper class to create an object containing the primitive value (Boolean, Byte, Character, Short, Integer, Long, Float or Double). Figure 20.2 provides a GUI that enables the user to test each of the Stack methods.

```
1 // Fig. 20.2: StackTest.java
2 // Testing the Stack class of the java.util package
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7 import java.util.*;
8
9 // Java extension packages
10 import javax.swing.*;
11
12 public class StackTest extends JFrame {
13     private JLabel statusLabel;
14     private JTextField inputField;
15     private Stack stack;
16
17     // create GUI to manipulate a Stack
18     public StackTest()
19     {
20         super( "Stacks" );
21     }
```

Fig. 20.2 Demonstrating class Stack of package java.util (part 1 of 5).
Container container = getContentPane();

statusLabel = new JLabel();
stack = new Stack();

container.setLayout( new FlowLayout() );
container.add( new JLabel( "Enter a string" ) );
inputField = new JTextField( 10 );
container.add( inputField );

// button to place object on stack
JButton pushButton = new JButton( "Push" );
pushButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            // put object on Stack
            statusLabel.setText( "Pushed: " +
                stack.push( inputField.getText() ) );
        }
    });
container.add( pushButton );

// button to remove top object on stack
JButton popButton = new JButton( "Pop" );
popButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event )
        {
            // remove element from Stack
            try {
                statusLabel.setText( "Popped: " + stack.pop() );
            }
            catch ( EmptyStackException exception ) {
                statusLabel.setText( exception.toString() );
            }
        }
    });
container.add( popButton );
// button to look at top element of stack
JButton peekButton = new JButton( "Peek" );

peekButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // look at top object on Stack
            try {
                statusLabel.setText( "Top: " + stack.peek() );
            }
            // process exception if Stack empty
            catch ( EmptyStackException exception ) {
                statusLabel.setText( exception.toString() );
            }
        }
    }
);  
container.add( peekButton );

// button to determine whether stack is empty
JButton emptyButton = new JButton( "Is Empty?" );

emptyButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // determine if Stack is empty
            statusLabel.setText( stack.empty() ?
                "Stack is empty" : "Stack is not empty" );
        }
    }
);  
container.add( emptyButton );

// button to determine whether search key is in stack
JButton searchButton = new JButton( "Search" );

searchButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // search Stack for specified object
            String searchKey = inputField.getText();
        }
    }
);  
container.add( searchButton );

Fig. 20.2 Demonstrating class Stack of package java.util (part 3 of 5).
int result = stack.search( searchKey );

if ( result == -1 )
    statusLabel.setText( searchKey + " not found" );
else
    statusLabel.setText( searchKey + " found at element " + result );

container.add( searchButton );

// button to display stack contents
JButton displayButton = new JButton( "Display" );
displayButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            // output Stack contents
            Enumeration enumeration = stack.elements();
            StringBuffer buffer = new StringBuffer();
            while ( enumeration.hasMoreElements() )
                buffer.append( enumeration.nextElement() ).append( " " );
            JOptionPane.showMessageDialog( null, buffer.toString(), "Display", JOptionPane.PLAIN_MESSAGE );
        }
    }
);

container.add( displayButton );
container.add( statusLabel );

setSize( 675, 100 );
setVisible( true );

// execute application
public static void main( String args[] ) {
    StackTest application = new StackTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}

Fig. 20.2 Demonstrating class Stack of package java.util (part 4 of 5).
1160 Java Utilities Package and Bit Manipulation Chapter 20

Line 25 creates an empty Stack. Line 43 calls Stack method push to add its argument to the top of the stack. The method returns an Object reference to its argument.

Line 61 calls Stack method pop to remove the top element of the stack. The method returns an Object reference to the removed element. If there are no elements in the Stack, method pop throws an EmptyStackException.

Line 85 calls Stack method peek to view the top element of the stack without removing the element. Method peek returns an Object reference to the element.

Line 108 calls Stack method empty to determine whether the stack is empty. If it is empty, the method returns true; otherwise, the method returns false.

Line 127 calls Stack method search to determine whether its argument is in the stack. If so, the method returns the position of the element in the stack. Note that the top element is position 1. If the element is not in the stack, -1 is returned.

The entire public interface of class Vector is actually part of class Stack, because Stack inherits from Vector. To prove this, our example provides a button to display the contents of the stack. This button invokes method elements to get an Enumeration of the stack; it then uses the Enumeration to walk through the stack elements.

Testing and Debugging Tip 20.1

Stack extends Vector, so the user may perform operations on Stack objects that are ordinarily not allowed on conventional stack data structures. This could "corrupt" the elements of the Stack and destroy the integrity of the Stack.

20.4 Dictionary Class

A Dictionary maps keys to values. When searching a Dictionary for a value, the program provides a key and the Dictionary returns the corresponding value. Dictionary is an abstract class. In particular, it is the superclass of class Hashtable, which we discuss in Section 20.5. Class Dictionary provides the public interface methods required to maintain a table of key–value pairs where the keys help store and retrieve the values in the table. Each key in the table is unique. The data structure is similar to a dictionary of words and definitions—the word is the key that is used to look up the definition (i.e., the value).

Dictionary method size returns the number of key–value pairs in a Dictionary object. Method isEmpty returns true if a Dictionary is empty, and false otherwise. Method keys returns an Enumeration that a program can use to iterate through a Dictionary’s keys. Method elements returns an Enumeration that a program can use to iterate through a Dictionary’s values. Method get returns the object that corresponds to a given key value. Method put puts an object associated with a given key into the table. Method remove removes an element corresponding to a given key and returns a reference to it.
Object-oriented programming languages facilitate creating new data types. When a program creates objects of new or existing types, the program then needs to manage those objects efficiently. This includes storing and retrieving objects. Storing and retrieving information with arrays is efficient if some aspect of your data directly matches the key value and if those keys are unique and tightly packed. If you have 100 employees with 9-digit Social Security numbers and you want to store and retrieve employee data by using the Social Security number as a key, it would nominally require an array with 999,999,999 elements, because there are 999,999,999 unique 9-digit numbers. This is impractical for virtually all applications that use Social Security numbers as keys. If the program could have an array that large, the program could get high performance for both storing and retrieving employee records by simply using the Social Security number as the array index.

There are numerous applications that have this problem, namely, that either the keys are of the wrong type (i.e., not nonnegative integers), or they may be of the right type, but sparsely spread over a huge range.

What is needed is a high-speed scheme for converting keys such as Social Security numbers, inventory part numbers and the like into unique array subscripts. Then, when an application needs to store something, the scheme could convert the application key rapidly into a subscript and the record of information could be stored at that slot in the array. Retrieval is accomplished the same way: Once the application has a key for which it wants to retrieve the data record, the application simply applies the conversion to the key—this produces the array subscript where the data is stored and the data is retrieved.

The scheme we describe here is the basis of a technique called hashing. Why the name? When we convert a key into an array subscript, we literally scramble the bits, forming a kind of “mishmashed” number. The number actually has no real significance beyond its usefulness in storing and retrieving this particular number data record.

A glitch in the scheme occurs when collisions occur (i.e., when two different keys “hash into” the same cell (or element) in the array). We cannot store two different data records in the same space, so we need to find an alternative home for all records beyond the first that hash to a particular array subscript. There are many schemes for doing this. One is to “hash again” (i.e., to reapply the hashing transformation to the key to provide a next candidate cell in the array). The hashing process is designed to distribute the values throughout the table, so the assumption is that with just a few hashes an available cell will be found.

Another scheme uses one hash to locate the first candidate cell. If that cell is occupied, successive cells are searched linearly until an available cell is found. Retrieval works the same way: The key is hashed once to determine the initial location to check to see whether it contains the desired data. If it does, the search is finished. If it does not, successive cells are searched linearly until the desired data is found.

The most popular solution to hash table collisions is to have each cell of the table be a hash “bucket,” typically a linked list of all the key–value pairs that hash to that cell. This is the solution that Java’s Hashtable class (from package java.util) implements.

One factor that affects the performance of hashing schemes is called the load factor. This is the ratio of the number of occupied cells in the hash table to the size of the hash table. The closer this ratio gets to 1.0, the greater the chance of collisions.
Performance Tip 20.6

The load factor in a hash table is a classic example of a space–time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower, due to increased hashing collisions. By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization, because a larger portion of the hash table remains empty.

The complexity of programming hash tables properly is too much for most casual programmers. Computer science students study hashing schemes thoroughly in courses called “Data Structures” or “Algorithms.” Recognizing the value of hashing to most programmers, Java provides class Hashtable and some related features to enable programmers to use hashing without having to implement the messy details.

Actually, the preceding sentence is profoundly important in our study of object-oriented programming. As discussed in earlier chapters, classes encapsulate and hide complexity (i.e., implementation details) and offer user-friendly interfaces. Crafting classes to do this properly is one of the most valued skills in the field of object-oriented programming.

Figure 20.3 provides a GUI that enables you to test several Hashtable methods. Line 25 creates an empty Hashtable with a default capacity of 101 elements and a default load factor of .75. When the number of occupied slots in the Hashtable becomes more than the capacity times the load factor, the table grows larger. Class Hashtable also provides a constructor that takes one argument specifying the capacity and a constructor that takes two arguments, specifying the capacity and load factor, respectively.

```java
// Fig. 20.3: HashtableTest.java
// Demonstrates class Hashtable of the java.util package.

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

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

public class HashtableTest extends JFrame {
    private JLabel statusLabel;
    private Hashtable table;
    private JTextArea displayArea;
    private JTextField lastNameField;
    private JTextField firstNameField;

    // set up GUI to demonstrate Hashtable features
    public HashtableTest() {
        super( "Hashtable Example" );
        statusLabel = new JLabel();
        table = new Hashtable();
        displayArea = new JTextArea( 4, 20 );
    }
}
```

Fig. 20.3 Demonstrating class Hashtable (part 1 of 6).
displayArea.setEditable( false );

JPanel northSubPanel = new JPanel();
northSubPanel.add( new JLabel( "First name" ) );
firstNameField = new JTextField( 8 );
northSubPanel.add( firstNameField );
northSubPanel.add( new JLabel( "Last name (key)" ) );
lastNameField = new JTextField( 8 );
northSubPanel.add( lastNameField );

JPanel northPanel = new JPanel();
northPanel.setLayout( new BorderLayout() );
northPanel.add( northSubPanel, BorderLayout.NORTH );
northPanel.add( statusLabel, BorderLayout.SOUTH );

JPanel southPanel = new JPanel();
southPanel.setLayout( new GridLayout( 2, 5 ) );
JButton putButton = new JButton( "Put" );
putButton.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent event ) {
        Employee employee = new Employee( 
            firstNameField.getText(), 
            lastNameField.getText() );

        Object value = 
            table.put( lastNameField.getText(), employee );

        // first time this key was added
        if ( value == null )
            statusLabel.setText( "Put: " + employee.toString() );
        else
            statusLabel.setText( "Put: " + employee.toString() + 
                                "; Replaced: " + value.toString() );
    }
});
southPanel.add( putButton );

// button to get value for specific key
JButton getButton = new JButton( "Get" );

Fig. 20.3 Demonstrating class Hashtable (part 2 of 6).
getButton.addActionListener(

    new ActionListener() {

        // get value for specific key
        public void actionPerformed( ActionEvent event )
        {
            Object value = table.get( lastNameField.getText() );

            // value found for key
            if ( value != null )
                statusLabel.setText( "Get: " + value.toString() );

            // value not found for key
            else
                statusLabel.setText( "Get: " + lastNameField.getText() + " not in table" );

        }

    });

southPanel.add( getButton );

// button to remove key/value pair from table
JButton removeButton = new JButton( "Remove" );

removeButton.addActionListener(

    new ActionListener() {

        // remove key/value pair
        public void actionPerformed( ActionEvent event )
        {
            Object value =
                table.remove( lastNameField.getText() );

            // key found
            if ( value != null )
                statusLabel.setText( "Remove: " + value.toString() );

            // key not found
            else
                statusLabel.setText( "Remove: " + lastNameField.getText() + " not in table" );

        }

    });

southPanel.add( removeButton );

Fig. 20.3 Demonstrating class Hashtable (part 3 of 6).
// button to determine whether hash table is empty
JButton emptyButton = new JButton( "Empty" );

emptyButton.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent event ) {
      statusLabel.setText( "Empty: " + table.isEmpty() );
   }
});
southPanel.add( emptyButton );

// button to determine whether hash table contains key
JButton containsKeyButton = new JButton( "Contains key" );

containsKeyButton.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent event ) {
      statusLabel.setText( "Contains key: " + table.containsKey( lastNameField.getText() ) );
   }
});
southPanel.add( containsKeyButton );

// button to clear all hash table contents
JButton clearButton = new JButton( "Clear table" );
clearButton.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent event ) {
      table.clear();
      statusLabel.setText( "Clear: Table is now empty" );
   }
});
southPanel.add( clearButton );

Fig. 20.3 Demonstrating class Hashtable (part 4 of 6).
// button to display hash table elements
JButton listElementsButton = new JButton( "List objects" );

listElementsButton.addActionListener( new ActionListener() {

    // display hash table elements
    public void actionPerformed( ActionEvent event )
    {
        StringBuffer buffer = new StringBuffer();

        for ( Enumeration enumeration = table.elements();
             enumeration.hasMoreElements(); )
            buffer.append( enumeration.nextElement() ).append( '
' );

        displayArea.setText( buffer.toString() );
    }
});
southPanel.add( listElementsButton );

// button to display hash table keys
JButton listKeysButton = new JButton( "List keys" );

listKeysButton.addActionListener( new ActionListener() {

    // display hash table KEYS
    public void actionPerformed( ActionEvent event )
    {
        StringBuffer buffer = new StringBuffer();

        for ( Enumeration enumeration = table.keys();
             enumeration.hasMoreElements(); )
            buffer.append( enumeration.nextElement() ).append( '
' );

        JOptionPane.showMessageDialog( null,
                                      buffer.toString(), "Display",
                                      JOptionPane.PLAIN_MESSAGE );
    }
});
southPanel.add( listKeysButton );

Container container = getContentPane();
container.add( northPanel, BorderLayout.NORTH );

Fig. 20.3 Demonstrating class HasTable (part 5 of 6).
container.add( new JScrollPane( displayArea ),
        BorderLayout.CENTER );
container.add( southPanel, BorderLayout.SOUTH );

    setSize( 540, 300 );
    setVisible( true );
}

// execute application
public static void main( String args[] )
{
    HashtableTest application = new HashtableTest();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
} // end class HashtableTest

// Employee class to represent first and last name
class Employee {
    private String first, last;

    // initialize an Employee
    public Employee( String firstName, String lastName )
    {
        first = firstName;
        last = lastName;
    }

    // convert Employee to String representation
    public String toString()
    {
        return first + " " + last;
    }
}
}  // end class Employee

Fig. 20.3 Demonstrating class Hashtable (part 6 of 6).
Lines 59–60 call Hashtable method put to add a key (the first argument) and a value (the second argument) into the Hashtable. Method put returns null if key has not been inserted in the Hashtable previously. Otherwise, method put returns the original value for that key in the Hashtable; this helps the program manage cases in which it intends to replace the value stored for a given key. If either the key or the value is null, a NullPointerException occurs.

Line 88 calls Hashtable method get to locate the value associated with the key specified as an argument. If the key is present in the table, get returns an Object reference to the corresponding value; otherwise, the method returns null.

Lines 116–117 call Hashtable method remove to remove a key–value pair from the table. The method returns a reference to the removed Object. If there is no value mapped to the specified key, the method returns null.

Line 144 calls Hashtable method isEmpty, which returns true if the Hashtable is empty; otherwise it returns false.

Line 162 calls Hashtable method containsKey to determine whether the key specified as an argument is in the Hashtable (i.e., a value is associated with that key). If so, the method returns true; otherwise, the method returns false. Class Hashtable also provides method contains to determine whether the Object specified as its argument is in the Hashtable.

Line 179 calls Hashtable method clear to empty the Hashtable contents. Line 199 calls Hashtable method elements to obtain an Enumeration of the values in the Hashtable. Line 223 calls Hashtable method keys to obtain an Enumeration of the keys in the Hashtable.

For more information on class Hashtable and its methods, see the online Java API documentation.

20.6 Properties Class

A Properties object is a persistent Hashtable object that normally stores key–value pairs of Strings—assuming that you use methods setProperty and getProperty to manipulate the table rather than Hashtable methods put and get. By persistent, we mean that the Hashtable object can be written to an output stream and directed to a file, and read back in through an input stream. In fact, most objects in Java can now be output and input with Java’s object serialization (see Chapter 16). The Properties class extends class Hashtable, so Properties objects have the methods we discussed in Fig. 20.3. The keys and values in a Properties object must be of type String. Class Properties provides some additional methods that are demonstrated in Fig. 20.4.

```java
// Fig. 20.4: PropertiesTest.java
// Demonstrates class Properties of the java.util package.

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

Fig. 20.4 Demonstrating class Properties (part 1 of 6).
public class PropertiesTest extends JFrame {
  private JLabel statusLabel;
  private Properties table;
  private JTextArea displayArea;
  private JTextField valueField, nameField;

  // set up GUI to test Properties table
  public PropertiesTest()
  {
    super( "Properties Test" );

    // create Properties table
    table = new Properties();

    Container container = getContentPane();

    // set up NORTH of window's BorderLayout
    JPanel northSubPanel = new JPanel();
    northSubPanel.add( new JLabel( "Property value" ) );
    valueField = new JTextField( 10 );
    northSubPanel.add( valueField );
    northSubPanel.add( new JLabel( "Property name (key)" ) );
    nameField = new JTextField( 10 );
    northSubPanel.add( nameField );

    JPanel northPanel = new JPanel();
    northPanel.setLayout( new BorderLayout() );
    northPanel.add( northSubPanel, BorderLayout.NORTH );
    statusLabel = new JLabel();
    northPanel.add( statusLabel, BorderLayout.SOUTH );

    container.add( northPanel, BorderLayout.NORTH );

    // set up CENTER of window's BorderLayout
    displayArea = new JTextArea( 4, 35 );
    container.add( new JScrollPane( displayArea ),
      BorderLayout.CENTER );

    // set up SOUTH of window's BorderLayout
    JPanel southPanel = new JPanel();
    southPanel.setLayout( new GridLayout( 1, 5 ) );

    // button to put a name/value pair in Properties table
    JButton putButton = new JButton( "Put" );
}
putButton.addActionListener(
    new ActionListener() {
        // put name/value pair in Properties table
        public void actionPerformed( ActionEvent event ) {
            Object value = table.setProperty(  
                nameField.getText(), valueField.getText() );
            
            if ( value == null )
                showstatus( "Put: " + nameField.getText() +  
                    " " + valueField.getText() );
            else
                showstatus( "Put: " + nameField.getText() +  
                    " " + valueField.getText() +  
                    "; Replaced: " + value.toString() );
        
            listProperties();
        }
    } ); // end call to addActionListener

southPanel.add( putButton );

// button to empty contents of Properties table
JButton clearButton = new JButton( "Clear" );

clearButton.addActionListener(
    new ActionListener() {
        // use method clear to empty table
        public void actionPerformed( ActionEvent event ) {
            table.clear();
            showstatus( "Table in memory cleared" );
            listProperties();
        }
    } ); // end call to addActionListener

southPanel.add( clearButton );

// button to get value of a property
JButton getPropertyButton = new JButton( "Get property" );

getPropertyButton.addActionListener(
    new ActionListener() {

Fig. 20.4  Demonstrating class Properties (part 3 of 6).
// use method getProperty to obtain a property value
public void actionPerformed( ActionEvent event )
{
    Object value = table.getProperty(
        nameField.getText() );

    if ( value != null )
        showstatus( "Get property: " +
            nameField.getText() + " " +
            value.toString() );

    else
        showstatus( "Get: " + nameField.getText() +
            " not in table" );

    listProperties();
}
); // end call to addActionListener
southPanel.add( getPropertyButton );

// button to contents of Properties table to file
JButton saveButton = new JButton( "Save" );

saveButton.addActionListener(
    new ActionListener() {
        // use method save to place contents in file
        public void actionPerformed( ActionEvent event )
        {
            // save contents of table
            try {
                FileOutputStream output =
                    new FileOutputStream( "props.dat" );

                table.store( output, "Sample Properties" );
                output.close();

                listProperties();
            }

            // process problems with file output
            catch( IOException ioException ) {
                ioException.printStackTrace();
            }
        }
    }); // end call to addActionListener
southPanel.add( saveButton );

Fig. 20.4 Demonstrating class Properties (part 4 of 6).
// button to load contents of Properties table from file
JButton loadButton = new JButton( "Load" );

loadButton.addActionListener(
    new ActionListener() {
        // use method load to read contents from file
        public void actionPerformed( ActionEvent event ) {
            // load contents of table
            try {
                FileInputStream input =
                    new FileInputStream( "props.dat" );
                table.load( input );
                input.close();
                listProperties();
            }
            // process problems with file input
            catch( IOException ioException ) {
                ioException.printStackTrace();
            }
        }
    }); // end call to addActionListener
southPanel.add( loadButton );
container.add( southPanel, BorderLayout.SOUTH );
setSize( 550, 225 ); setVisible( true );

// output property values
public void listProperties() {
    StringBuffer buffer = new StringBuffer();
    String name, value;
    Enumeration enumeration = table.propertyNames();
    while ( enumeration.hasMoreElements() ) {
        name = enumeration.nextElement().toString();
        value = table.getProperty( name );
        buffer.append( name ).append( ' ' );
        buffer.append( value ).append( '
' );
    }
displayArea.setText( buffer.toString() );

Fig. 20.4 Demonstrating class Properties (part 5 of 6).
Chapter 20  Java Utilities Package and Bit Manipulation  1173

219  }  // end method ListProperties
220
221  // display String in statusLabel label
222  public void showstatus( String s )
223  {
224      statusLabel.setText( s );
225  }
226
227  // execute application
228  public static void main( String args[] )
229  {
230      PropertiesTest application = new PropertiesTest();
231
232      application.setDefaultCloseOperation(
233          JFrame.EXIT_ON_CLOSE );
234  }
235
236  }  // end class PropertiesTest

Fig. 20.4  Demonstrating class Properties (part 6 of 6).

Line 25 uses the no-argument constructor to create an empty Properties table with no default properties. Class Properties also provides an overloaded constructor that receives a reference to a Properties object containing default property values.

Lines 68–69 call Properties method setProperty to store a value for the specified key. If the key does not exist in the table, setProperty returns null; otherwise, it returns the previous value for that key.

Lines 116–117 call Properties method getProperty to locate the value associated with the specified key. If the key is not found in this Properties object, getProperty uses the one in the default Properties object (if there is one). The process continues recursively until there are no more default Properties objects (remember that every Properties object can be initialized with a default Properties object), at which point getProperty returns null. An overloaded version of this method receives two arguments, the second of which is the default value to return if getProperty cannot locate the key.

Line 150 calls Properties method store to save the contents of the Properties object to the OutputStream object specified as the first argument (in this case, a FileOutputStream). The String argument is a description of the Properties object. Class Properties also provides method list, which takes a PrintStream argument. This method is useful for displaying the set of properties.
Testing and Debugging Tip 20.2

Use `Properties` method `list` to display the contents of a `Properties` object for debugging purposes.

Line 181 calls `Properties` method `load` to restore the contents of the `Properties` object from the `InputStream` specified as the first argument (in this case, a `FileInputStream`).

Line 208 calls `Properties` method `propertyNames` to obtain an `Enumeration` of the property names. The value of each property can be determined by using method `getProperty`.

20.7 Random Class

We discussed random-number generation in Chapter 6, Methods, where we used `Math` class method `random`. Java provides extensive additional random number generation capabilities in class `Random`. We briefly walk through the API calls here.

A new random-number generator can be created by using

```java
Random r = new Random();
```

This form uses the computer’s current time to seed its random-number generator differently during each constructor call and thus generates different sequences of random numbers for each `Random` object.

To create a pseudorandom-number generator with “repeatability,” use

```java
Random r = new Random( seedValue );
```

The `seedValue` argument (type `long`) is used in the random number calculation to “seed” the random number generator. If the same `seedValue` is used every time, the `Random` object produces the same sequence of random numbers.

Testing and Debugging Tip 20.3

While a program is under development, use the form `Random(seedValue)` that produces a repeatable sequence of random numbers. If a bug occurs, fix the bug and test with the same `seedValue`; this allows you to reconstruct the exact same sequence of random numbers that caused the bug. Once the bugs have been removed, use the form `Random()`, which generates a new sequence of random numbers each time the program is run.

The call

```java
r.setSeed( seedValue );
```

resets `r`’s seed value at any time.

The calls

```java
r.nextInt()
r.nextLong()
```

generate uniformly distributed random integers. You can use `Math.abs` to take the absolute value of the number produced by `nextInt`, thus giving a number in the range from zero through approximately 2 billion. Then use the `%` operator to scale the number. For example, to roll a six-sided die, if you scale with a 6, you will get a number in the range from
0 through 5. Then simply shift this value by adding 1 to produce a number in the range from 
1 through 6. The expression is as follows:

\[
\text{Math.abs( r.nextInt() ) } \mod 6 + 1
\]

The calls

\[
\text{r.nextFloat()}
\]
\[
\text{r.nextDouble()}
\]
generate uniformly distributed values in the range 0.0 <= x < 1.0.

The call

\[
\text{r.nextGaussian()}
\]
generates a double value with a probability density of a Gaussian (i.e., “normal”) distri-
bution (mean of 0.0 and standard deviation of 1.0).

## 20.8 Bit Manipulation and the Bitwise Operators

Java provides extensive bit-manipulation capabilities for programmers who need to get 
down to the so-called “bits-and-bytes” level. Operating systems, test equipment software, 
networking software and many other kinds of software require that the programmer com-
unicate “directly with the hardware.” In this section and the next, we discuss Java’s bit-
manipulation capabilities. We introduce Java’s bitwise operators, and we demonstrate their 
use in live-code examples.

Computers represent all data internally as sequences of bits. Each bit can assume the 
value 0 or the value 1. On most systems, a sequence of 8 bits forms a byte—the standard 
storage unit for a variable of type byte. Other data types are stored in larger numbers of 
bytes. The bitwise operators can manipulate the bits of integral operands (i.e., those having 
type byte, char, short, int and long).

Note that the bitwise operator discussions in this section show the binary representa-
tions of the integer operands. For a detailed explanation of the binary (also called base 2) 
number system, see Appendix E, Number Systems.

The bitwise operators are bitwise AND (&), bitwise inclusive OR (|), bitwise exclusive 
OR (^), left shift (<<), right shift with sign extension (>>), right shift with zero extension 
(>>>), and complement (~). The bitwise AND, bitwise inclusive OR and bitwise exclusive 
OR operators compare their two operands bit by bit. The bitwise AND operator sets each 
bit in the result to 1 if the corresponding bit in both operands is 1. The bitwise inclusive OR 
operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) 
is 1. The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding 
bit in exactly one operand is 1. The left shift operator shifts the bits of its left operand to the 
left by the number of bits specified in its right operand. The right shift operator with sign 
extension shifts the bits in its left operand to the right by the number of bits specified in its 
right operand—if the left operand is negative, 1s are shifted in from the left; otherwise, 0s 
are shifted in from the left. The right shift operator with zero extension shifts the bits in its 
left operand to the right by the number of bits specified in its right operand—0s are shifted 
in from the left. The bitwise complement operator sets all 0 bits in its operand to 1 in the 
result and sets all 1 bits to 0 in the result. Detailed discussions of each bitwise operator 
appear in the following examples. The bitwise operators are summarized in Fig. 20.5.
When using the bitwise operators, it is useful to display values in their binary representation to illustrate the effects of these operators. The application of Fig. 20.6 allows the user to enter an integer into a JTextField and press Enter. Method actionPerformed (lines 32–36) reads the String from the JTextField, converts it to an integer and invokes method getBits (lines 55–80) to obtain a String representation of the integer in bits. The result is displayed in the output JTextField. The integer is displayed in its binary representation in groups of eight bits each. Method getBits uses the bitwise AND operator to combine variable value with variable displayMask. Often, the bitwise AND operator is used with an operand called a mask—an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits. In getBits, mask variable displayMask is assigned the value 1 << 31 or

10000000 00000000 00000000 00000000

The left shift operator shifts the value 1 from the low-order (rightmost) bit to the high-order (leftmost) bit in displayMask and fills in 0 bits from the right.

```java
1   // Fig. 20.6: PrintBits.java
2   // Printing an unsigned integer in bits
3
4   // Java core packages
5   import java.awt.*;
6   import java.awt.event.*;
```
```java
// Java extension packages
import javax.swing.*;

public class PrintBits extends JFrame {
  private JTextField outputField;

  // set up GUI
  public PrintBits() {
    super( "Printing bit representations for numbers" );
    Container container = getContentPane();
    container.setLayout( new FlowLayout() );
    container.add( new JLabel( "Enter an integer " ) );
    JTextField inputField = new JTextField( 10 );
    inputField.addActionListener( new ActionListener() {
      // read integer and get bitwise representation
      public void actionPerformed( ActionEvent event ) {
        int value = Integer.parseInt( event.getActionCommand() );
        outputField.setText( getBits( value ) );
      }
    } );
    container.add( inputField );
    container.add( new JLabel( "The integer in bits is" ) );
    JTextField outputField = new JTextField( 33 );
    outputField.setEditable( false );
    container.add( outputField );
    setSize( 720, 70 );
    setVisible( true );
  }

  // display bit representation of specified int value
  private String getBits( int value ) {
    // create int value with 1 in leftmost bit and 0s elsewhere
    int displayMask = 1 << 31;
  }
}
```

Fig. 20.6 Printing the bits in an integer (part 2 of 3).
Lines 68–69 append a 1 or a 0 to a `StringBuffer` for the current leftmost bit of variable `value`. Assume that `value` contains `4000000000 (11101110 01101011 00101000 00000000)`. When `value` and `displayMask` are combined using `&`, all the bits except the high-order (leftmost) bit in variable `value` are “masked off” (hidden), because any bit “ANDed” with 0 yields 0. If the leftmost bit is 1, `value & displayMask` evaluates to a nonzero value and 1 is appended; otherwise, 0 is appended. Then vari-
able value is left shifted one bit by the expression value <<= 1 (this is equivalent to value = value << 1). These steps are repeated for each bit in variable value. At the end of method getBits, the StringBuffer is converted to a String in line 79 and returned from the method. Figure 20.7 summarizes the results of combining two bits with the bitwise AND (&) operator.

**Common Programming Error 20.1**

Using the logical AND operator (&&) for the bitwise AND operator (&) is a common programming error.

Figure 20.8 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator. The program uses method getBits (lines 163–188) to get a String representation of the integer values. The program allows the user to enter values into JTextField (for the binary operators, two values must be entered), and then to press the button representing the operation they would like to test. The program displays the result of each operation in both integer and bitwise representations.

The first output window for Fig. 20.8 shows the results of combining the value 65535 and the value 1 with the bitwise AND operator (&). All the bits except the low-order bit in the value 65535 are “masked off” (hidden) by “ANDing” with the value 1.

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 1 &amp; Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Fig. 20.7** Results of combining two bits with the bitwise AND operator (&).

```java
// Fig. 20.8: MiscBitOps.java
// Using the bitwise AND, bitwise inclusive OR, bitwise
// exclusive OR, and bitwise complement operators.

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

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

public class MiscBitOps extends JFrame {
    private JTextField input1Field, input2Field,
        bits1Field, bits2Field, bits3Field, resultField;
```
private int value1, value2;

// set up GUI
public MiscBitOps()
{
    super( "Bitwise operators" );

    JPanel inputPanel = new JPanel();
    inputPanel.setLayout( new GridLayout( 4, 2 ) );

    inputPanel.add( new JLabel( "Enter 2 ints" ) );
    inputPanel.add( new JLabel( "" ) );

    inputPanel.add( new JLabel( "Value 1" ) );
    input1Field = new JTextField( 8 );
    inputPanel.add( input1Field );

    inputPanel.add( new JLabel( "Value 2" ) );
    input2Field = new JTextField( 8 );
    inputPanel.add( input2Field );

    inputPanel.add( new JLabel( "Result" ) );
    resultField = new JTextField( 8 );
    resultField.setEditable( false );
    inputPanel.add( resultField );

    JPanel bitsPanel = new JPanel();
    bitsPanel.setLayout( new GridLayout( 4, 1 ) );
    bitsPanel.add( new JLabel( "Bit representations" ) );

    bits1Field = new JTextField( 33 );
    bits1Field.setEditable( false );
    bitsPanel.add( bits1Field );

    bits2Field = new JTextField( 33 );
    bits2Field.setEditable( false );
    bitsPanel.add( bits2Field );

    bits3Field = new JTextField( 33 );
    bits3Field.setEditable( false );
    bitsPanel.add( bits3Field );

    JPanel buttonPanel = new JPanel();

    // button to perform bitwise AND
    JButton andButton = new JButton( "AND" );

    andButton.addActionListener(
        new ActionListener() {
            new ActionListener();
        }
    );
// perform bitwise AND and display results
public void actionPerformed(ActionEvent event)
{
    setFields();
    resultField.setText(
        Integer.toString(value1 & value2));
    bits3Field.setText(getBits(value1 & value2));
}

buttonPanel.add(andButton);

// button to perform bitwise inclusive OR
JButton inclusiveOrButton = new JButton("Inclusive OR");
inclusiveOrButton.addActionListener(

    new ActionListener()
    {
        // perform bitwise inclusive OR and display results
        public void actionPerformed(ActionEvent event)
        {
            setFields();
            resultField.setText(
                Integer.toString(value1 | value2));
            bits3Field.setText(getBits(value1 | value2));
        }
    }
);

buttonPanel.add(inclusiveOrButton);

// button to perform bitwise exclusive OR
JButton exclusiveOrButton = new JButton("Exclusive OR");
exclusiveOrButton.addActionListener(

    new ActionListener()
    {
        // perform bitwise exclusive OR and display results
        public void actionPerformed(ActionEvent event)
        {
            setFields();
            resultField.setText(
                Integer.toString(value1 ^ value2));
            bits3Field.setText(getBits(value1 ^ value2));
        }
    }
);

buttonPanel.add(exclusiveOrButton);

Fig. 20.8 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators (part 3 of 6).
// button to perform bitwise complement
JButton complementButton = new JButton( "Complement" );

complementButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent event ) {
            input2Field.setText( "" );
            bits2Field.setText( "" );
            int value = Integer.parseInt( input1Field.getText() );
            resultField.setText( Integer.toString( ~value ) );
            bits1Field.setText( getBits( value ) );
            bits3Field.setText( getBits( ~value ) );
        }
    });

buttonPanel.add( complementButton );

Container container = getContentPane();
container.add( inputPanel, BorderLayout.WEST );
container.add( bitsPanel, BorderLayout.EAST );
container.add( buttonPanel, BorderLayout.SOUTH );

setSize( 600, 150 );
setVisible( true );

// display numbers and their bit form
private void setFields()
{
    value1 = Integer.parseInt( input1Field.getText() );
    value2 = Integer.parseInt( input2Field.getText() );
    bits1Field.setText( getBits( value1 ) );
    bits2Field.setText( getBits( value2 ) );
}

// display bit representation of specified int value
private String getBits( int value )
{
    // create int value with 1 in leftmost bit and 0s elsewhere
    int displayMask = 1 << 31;
    // buffer to build output
    StringBuffer buffer = new StringBuffer( 35 );
for ( int bit = 1; bit <= 32; bit++ ) {
    // use displayMask to isolate bit and determine whether
    // bit has value of 0 or 1
    buffer.append(( value & displayMask ) == 0 ? '0' : '1 ');

    // shift value one position to left
    value <<= 1;

    // append space to buffer every 8 bits
    if ( bit % 8 == 0 )
        buffer.append( ' ' );
}

return buffer.toString();

// execute application
public static void main( String args[] )
{
    MiscBitOps application = new MiscBitOps();

    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
}  // end class MiscBitOps

Fig. 20.8 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators (part 5 of 6).
The bitwise inclusive OR operator sets specific bits to 1 in an operand. The second output window for Fig. 20.8 shows the results of combining the value 15 and the value 241 by using the bitwise OR operator—the result is 255. Figure 20.9 summarizes the results of combining two bits with the bitwise inclusive OR operator.

**Common Programming Error 20.2**

*Using the logical OR operator (||) for the bitwise OR operator (|) is a common programming error.*

The bitwise exclusive OR operator (^) sets each bit in the result to 1 if *exactly* one of the corresponding bits in its two operands is 1. The third output of Fig. 20.8 shows the results of combining the value 139 and the value 199 by using the exclusive OR operator—the result is 76. Figure 20.10 summarizes the results of combining two bits with the bitwise exclusive OR operator.
The bitwise complement operator (\(~\)) sets all 1 bits in its operand to 0 in the result and sets all 0 bits to 1 in the result—otherwise referred to as “taking the one’s complement of the value.” The fourth output window for Fig. 20.8 shows the results of taking the one’s complement of the value 21845. The result is -21846.

The program of Fig. 20.11 demonstrates the left shift operator (\(<<\)), the right shift operator with sign extension (\(>>\)) and the right shift operator with zero extension (\(>>>\)). Method `getBits` (lines 113–138) obtains a `String` containing the bit representation of the integer values. The program allows the user to enter an integer into a `JTextField` and press `Enter` to display the bit representation of the integer in a second `JTextField`. The the user can press a button representing a shift operation to perform a 1-bit shift and view the results of the shift in both integer and bitwise representation.

```java
1 // Fig. 20.11: BitShift.java
2 // Using the bitwise shift operators.
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Java extension packages
9 import javax.swing.*;
10
11 public class BitShift extends JFrame {
12    private JTextField bitsField;
13    private JTextField valueField;
14
15    // set up GUI
16    public BitShift() {
17        super( "Shifting bits" );
18
19        Container container = getContentPane();
20        container.setLayout( new FlowLayout() );
21        container.add( new JLabel( "Integer to shift " ) );
22        // textfield for user to input integer
23        valueField = new JTextField( 12 );
```

Fig. 20.10 Results of combining two bits with the bitwise exclusive OR operator (\(^\)).

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 1 ^ Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 20.11 Demonstrating the bitwise shift operators (part 1 of 5).
container.add( valueField );

valueField.addActionListener(
        new ActionListener() {
            // read value and display its bitwise representation
            public void actionPerformed( ActionEvent event )
            {
                int value = Integer.parseInt( valueField.getText() );
                bitsField.setText( getBits( value ) );
            }
        });

    // textfield to display bitwise representation of an integer
    bitsField = new JTextField( 33 );
    bitsField.setEditable( false );
    container.add( bitsField );

    // button to shift bits left by one position
    JButton leftButton = new JButton( "<<" );

    leftButton.addActionListener(
            new ActionListener() {
                // left shift one position and display new value
                public void actionPerformed( ActionEvent event )
                {
                    int value = Integer.parseInt( valueField.getText() );
                    value <<= 1;
                    valueField.setText( Integer.toString( value ) );
                    bitsField.setText( getBits( value ) );
                }
            });
    container.add( leftButton );

    // button to right shift value one position with sign extension
    JButton rightSignButton = new JButton( ">>" );

    rightSignButton.addActionListener(
            new ActionListener() {
                // right shift one position and display new value
                public void actionPerformed( ActionEvent event )
                {
                    int value = Integer.parseInt( valueField.getText() );
                    value >>= 1;
                    valueField.setText( Integer.toString( value ) );
                }
            });
bitsField.setText( getBits( value ) );
}
);

container.add( rightSignButton );

// button to right shift value one position with zero extension
JButton rightZeroButton = new JButton( ">>>" );

rightZeroButton.addActionListener(  
    new ActionListener() {  
        // right shift one position and display new value  
        public void actionPerformed( ActionEvent event ) {  
            int value = Integer.parseInt( valueField.getText() );  
            value >>>= 1;
            valueField.setText( Integer.toString( value ) );
            bitsField.setText( getBits( value ) );
        }
    });

container.add( rightZeroButton );

setSize( 400, 120 );
setVisible( true );

// display bit representation of specified int value
private String getBits( int value ) {
    // create int value with 1 in leftmost bit and 0s elsewhere
    int displayMask = 1 << 31;

    // buffer to build output
    StringBuffer buffer = new StringBuffer( 35 );

    // for each bit append 0 or 1 to buffer
    for ( int bit = 1; bit <= 32; bit++ ) {
        // use displayMask to isolate bit and determine whether
        // bit has value of 0 or 1
        buffer.append(    ( value & displayMask ) == 0 ? '0' : '1' );

        // shift value one position to left
        value <<= 1;
    }

    return buffer.toString();
}

Fig. 20.11 Demonstrating the bitwise shift operators (part 3 of 5).
// append space to buffer every 8 bits
if ( bit % 8 == 0 )
    buffer.append( ' ' );
}
return buffer.toString();

// execute application
public static void main( String args[] )
{
    BitShift application = new BitShift();
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
}
} // end class BitShift

Fig. 20.11 Demonstrating the bitwise shift operators (part 4 of 5).
The left shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand (performed at line 58 in the program). Bits vacated to the right are replaced with 0s; 1s shifted off the left are lost. The first four output windows of Fig. 20.11 demonstrate the left shift operator. Starting with the value 1, the left shift button was pressed twice, resulting in the values 2 and 4, respectively. The fourth output window shows the result of value1 being shifted 31 times. Note that the result is a negative value. That is because a 1 in the high-order bit is used to indicate a negative value in an integer.

The right shift operator with sign extension (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand (performed at line 78 in the program). Performing a right shift causes the vacated bits at the left to be replaced by 0s if the number is positive or 1s if the number is negative. Any 1s shifted off the right are lost. The fifth and sixth output windows show the results of right shifting (with sign extension) the value in the fourth output window two times.

The right shift operator with zero extension (>>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand (performed at line 98 in the program). Performing a right shift causes the vacated bits at the left to be replaced by 0s. Any 1s shifted off the right are lost. The eighth and ninth output windows show the results of right shifting (with zero extension) the value in the seventh output window two times.

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These bitwise assignment operators are shown in Fig. 20.12.
Class `BitSet` makes it easy to create and manipulate bit sets. Bit sets are useful for representing a set of boolean flags. `BitSets` are dynamically resizable. More bits can be added as needed, and a `BitSet` object will grow to accommodate the additional bits. The statement

```java
BitSet b = new BitSet();
```

creates a `BitSet` that initially is empty. Also, a program can specify the size of a `BitSet` with the statement

```java
BitSet b = new BitSet( size );
```

which creates a `BitSet` with `size` bits.

The statement

```java
b.set( bitNumber );
```

sets bit `bitNumber` “on.” This makes the underlying value of that bit 1. Note that bit numbers are zero based, like `Vectors`. The statement

```java
b.clear( bitNumber );
```

sets bit `bitNumber` “off.” This makes the underlying value of that bit 0. The statement

```java
b.get( bitNumber );
```

gets the value of bit `bitNumber`. The result is returned as `true` if the bit is on, `false` if the bit is off.

The statement

```java
b.and( b1 );
```

performs a bit-by-bit logical AND between `BitSets b` and `b1`. The result is stored in `b`. Bitwise logical OR and bitwise logical XOR are performed by the statements

```java
b.or( b1 );
b.xor( b2 );
```
The expression

\[ b.size() \]

returns the size of the `BitSet`. The expression

\[ b.equals( b1 ) \]

compares the two `BitSet`s for equality. The expression

\[ b.toString() \]

creates a `String` representation of the `BitSet` contents. This is helpful for debugging.

Figure 20.13 revisits the Sieve of Eratosthenes for finding prime numbers, which we discussed in Exercise 7.27. This example uses a `BitSet` rather than an array to implement the algorithm. The program displays all the prime numbers from 2 to 1023 in a `JTextArea` and provides a `JTextField` in which the user can type any number from 2 to 1023 to determine whether that number is prime (in which case a message is displayed in a `JLabel`).

```java
// Fig. 20.13: BitSetTest.java
// Using a BitSet to demonstrate the Sieve of Eratosthenes.

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

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

public class BitSetTest extends JFrame {

private BitSet sieve;
private JLabel statusLabel;
private JTextField inputField;

// set up GUI
public BitSetTest()
{
    super( "BitSets" );

    sieve = new BitSet( 1024 );

    Container container = getContentPane();

    statusLabel = new JLabel( "" );
    container.add( statusLabel, BorderLayout.SOUTH );

    JPanel inputPanel = new JPanel();

    inputPanel.add( new JLabel( "Enter a value from 2 to 1023" ) );
}
```

Fig. 20.13 Demonstrating the Sieve of Eratosthenes using a `BitSet` (part 1 of 3).
// textfield for user to input a value from 2 to 1023
inputField = new JTextField( 10 );

inputField.addActionListener(
    new ActionListener() {
        // determine whether value is prime number
        public void actionPerformed( ActionEvent event )
        {
            int value = Integer.parseInt( inputField.getText() );

            if ( sieve.get( value ) )
                statusLabel.setText( value + " is a prime number" );
            else
                statusLabel.setText( value + " is not a prime number" );
        }
    });

inputPanel.add( inputField );
container.add( inputPanel, BorderLayout.NORTH );

JTextArea primesArea = new JTextArea();
container.add( new JScrollPane( primesArea ), BorderLayout.CENTER );

// set all bits from 1 to 1023
int size = sieve.size();

for ( int i = 2; i < size; i++ )
    sieve.set( i );

// perform Sieve of Eratosthenes
int finalBit = ( int ) Math.sqrt( sieve.size() );

for ( int i = 2; i < finalBit; i++ )
    if ( sieve.get( i ) )
        for ( int j = 2 * i; j < size; j += i )
            sieve.clear( j );

// display prime numbers from 1 to 1023
int counter = 0;

Fig. 20.13 Demonstrating the Sieve of Eratosthenes using a BitSet (part 2 of 3).
Line 22 creates a `BitSet` of 1024 bits. We ignore the bit at index 0 in this program. Lines 68–69 set all the bits in the `BitSet` to on with `BitSet` method `set`. Lines 72–79 determine all the prime numbers from 2 to 1023. The integer `finalBit` specifies when the algorithm is complete. The basic algorithm is that a number is prime if it has no divisors other than 1 and itself. Starting with the number 2, once we know a number is prime, we can eliminate all multiples of that number. The number 2 is only divisible by 1 and itself, so it is prime. Therefore, we can eliminate 4, 6, 8 and so on. Elimination of a value consists...
of setting its bit to off with BitSet method clear. The number 3 is divisible by 1 and itself. Therefore, we can eliminate all multiples of 3 (keep in mind that all even numbers have already been eliminated). After the list of primes is displayed, the user can type a value from 2 to 1023 in the textfield and press enter to determine whether the number is prime. Method actionPerformed (lines 42–53) uses BitSet method get (line 46) to determine whether the bit for the number the user entered is set. If so, lines 47–48 display a message indicating that the number is prime. Otherwise, lines 51–52 display a message indicating that the number is not prime.

**SUMMARY**

- Class Vector manages dynamically resizable arrays. At any time the Vector contains a certain number of elements which is less than or equal to its capacity. The capacity is the space that has been reserved for the array.

- If a Vector needs to grow, it grows by an increment that you specify or by a default assumed by the system. If you do not specify a capacity increment, the system automatically doubles the size of the Vector each time additional capacity is required.

- Vectors store references to Objects. To store values of primitive data types in Vectors, use the type-wrapper classes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` and `Character`) to create objects containing the primitive data type values.

- Class Vector provides three constructors. The no-argument constructor creates an empty Vector. The constructor that takes one argument creates a Vector with an initial capacity specified by the argument. The constructor that takes two arguments creates a Vector with an initial capacity specified by the first argument and a capacity increment specified by the second argument.

- Vector method addElement adds its argument to the end of the Vector. Method insertElementAt inserts an element at the specified position. Method setElementAt sets the element at a specific position.

- Vector method removeElement removes the first occurrence of its argument. Method removeAllElements removes every element from the Vector. Method removeElementAt removes the element at the specified index.

- Vector method firstElement returns a reference to the first element. Method lastElement returns a reference to the last element.

- Vector method isEmpty determines whether the Vector is empty.

- Vector method contains determines whether the Vector contains the searchKey specified as an argument.

- Vector method indexOf gets the index of the first location of its argument. The method returns −1 if the argument is not found in the Vector.

- Vector method trimToSize cuts the capacity of the Vector to the Vector’s size. Methods size and capacity determine the number of elements currently in the Vector and the number of elements that can be stored in the Vector without allocating more memory, respectively.

- Vector method elements returns a reference to an Enumeration containing the elements of the Vector.

- Enumeration method hasMoreElements determines whether there are more elements. Method nextElement returns a reference to the next element.

- Class Stack extends class Vector. Stack method push adds its argument to the top of the stack. Method pop removes the top element of the stack. Method peek returns an Object ref-
erence to the top element of the stack without removing the element. **Stack** method **empty** determines whether the stack is empty.

- A **Dictionary** transforms keys to values.
- Hashing is a high-speed scheme for converting keys into unique array subscripts for storage and retrieval of information. The load factor is the ratio of the number of occupied cells in a hash table to the size of the hash table. The closer this ratio gets to 1.0, the greater the chance of collisions.
- The no-argument **Hashtable** constructor creates a **Hashtable** with a default capacity of 101 elements and a default load factor of .75. The **Hashtable** constructor that takes one argument specifies the initial capacity; the constructor that takes two arguments specifies the initial capacity and load factor, respectively.
- **Hashtable** method **put** adds a key and a value into a **Hashtable**. Method **get** locates the value associated with the specified key. Method **remove** deletes the value associated with the specified key. Method **isEmpty** determines whether the table is empty.
- **Hashtable** method **containsKey** determines whether the key specified as an argument is in the **Hashtable** (i.e., a value is associated with that key). Method **contains** determines whether the **Object** specified as its argument is in the **Hashtable**. Method **clear** empties the **Hashtable**. Method **elements** obtains an **Enumeration** of the values. Method **keys** obtains an **Enumeration** of the keys.
- A **Properties** object is a persistent **Hashtable** object. Class **Properties** extends **Hashtable**. Keys and values in a **Properties** object must be **Strings**.
- The **Properties** no-argument constructor creates an empty **Properties** table with no default properties. There is also an overloaded constructor that is passed a reference to a default **Properties** object containing default property values.
- **Properties** method **getProperty** locates the value of the key specified as an argument. Method **store** saves the contents of the **Properties** object to the **OutputStream** object specified as the first argument. Method **load** restores the contents of the **Properties** object from the **InputStream** object specified as the argument. Method **propertyNames** obtains an **Enumeration** of the property names.
- Java provides extensive random-number generation capabilities in class **Random**. Class **Random**’s no-argument constructor uses the time to seed its random-number generator differently each time it is called. To create a pseudorandom-number generator with repeatability, use the **Random** constructor that takes a seed argument.
- **Random** method **setSeed** sets the seed. Methods **nextInt** and **nextLong** generate uniformly distributed random integers. Methods **nextFloat** and **nextDouble** generate uniformly distributed values in the range $0.0 \leq x < 1.0$.
- The bitwise AND (&) operator sets each bit in the result to 1 if the corresponding bit in both operands is 1.
- The bitwise inclusive OR (|) operator sets each bit in the result to 1 if the corresponding bit in either (or both) operand(s) is 1.
- The bitwise exclusive OR (^) operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1.
- The left shift (<<) operator shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- The right shift operator with sign extension (>>>) shifts the bits in its left operand to the right by the number of bits specified in its right operand—if the left operand is negative, 1s are shifted in from the left; otherwise, 0s are shifted in from the left.
Java Utilities Package and Bit Manipulation

Chapter 20

- The right shift operator with zero extension (\( \gg\gg\gg \)) shifts the bits in its left operand to the right by the number of bits specified in its right operand—\( 0 \)'s are shifted in from the left.
- The bitwise complement (\( \sim \)) operator sets all \( 0 \) bits in its operand to \( 1 \) in the result and sets all \( 1 \) bits to \( 0 \) in the result.
- Each bitwise operator (except complement) has a corresponding assignment operator.
- The no-argument `BitSet` constructor creates an empty `BitSet`. The one-argument `BitSet` constructor creates a `BitSet` with the number of bits specified by its argument.
- `BitSet` method `set` sets the specified bit “on.” Method `clear` sets the specified bit “off.” Method `get` returns `true` if the bit is on, `false` if the bit is off.
- `BitSet` method `and` performs a bit-by-bit logical AND between `BitSets`. The result is stored in the `BitSet` that invoked the method. Similarly, bitwise logical OR and bitwise logical XOR are performed by methods `or` and `xor`.
- `BitSet` method `size` returns the size of a `BitSet`. Method `toString` converts a `BitSet` to a `String`.

**TERMINOLOGY**

- `addElement` method of class `Vector`:
  - bit set

- `and` method of class `BitSet`:
  - bitwise assignment operators
    - `&=` (bitwise AND)
    - `^=` (bitwise exclusive OR)
    - `|=` (bitwise inclusive OR)
    - `<<=` (left shift)
    - `>>=` (right shift)
    - `>>>=` (right shift with zero extension)

- `BitSet` class:
  - bitwise manipulation operators
    - `&` bitwise AND
    - `^` bitwise exclusive OR
    - `|` bitwise inclusive OR
    - `~` one’s complement
    - `<<` left shift
    - `>>` right shift
    - `>>>` right shift with zero extension

- `capacity` method of class `Vector`:
  - capacity increment of a `Vector`

- `clear` method of class `BitSet`:

- `clear` method of class `Hashtable`:

- `clone` method of class `BitSet`:

- `contains` method of class `Vector`:

- `containsKey` method of class `Hashtable`:

- `Dictionary` class:

- `dynamically resizable array`:

- `elementAt` method of class `Vector`:

- `elements` method of class `Dictionary`:

- `elements` method of class `Vector`:

- `EmptyStackException` class:

- `enumerate successive elements`:

- `Enumeration` interface:

- `equals` method of class `Object`:

- `firstElement` method of class `Vector`:

- `get` method of class `BitSet`:

- `get` method of class `Dictionary`:

- `getPropertyName` method of class `Properties`:

- `hashCode` method of class `Object`:

- `Hashtable` class:

- `hasMoreElements` method (Enumeration):

- `indexOf` method of class `Vector`:

- `insertElementAt` method of class `Vector`:

- `isEmpty` method of class `Dictionary`:

- `isEmpty` method of class `Vector`:

- `iterate through container elements`:

- `java.util` package:

- `key` in a `Dictionary`:

- `key/value pair`:

- `keys` method of class `Dictionary`:

- `lastElement` method of class `Vector`:

- `list method of class Properties`:

- `loadFactor in hashing`:

- `load` method of class `Properties`:

- `nextDouble` method of class `Random`:

- `nextElement` method of `Enumeration`:

- `nextFloat` method of `Random`:

- `nextInt` method of `Random`:
Chapter 20  Java Utilities Package and Bit Manipulation 1197

SELF-REVIEW EXERCISES

20.1 Fill in the blanks in each of the following statements:

a) Java class _________ provides the capabilities of array-like data structures that can resize themselves dynamically.

b) If you do not specify a capacity increment, the system will _________ the size of the Vector each time additional capacity is needed.

c) If storage is at a premium, use the _________ method of the Vector class to trim a Vector to its exact size.

20.2 State whether each of the following is true or false. If false, explain why.

a) Values of primitive data types may be stored directly in a Vector. 

b) With hashing, as the load factor increases, the chance of collisions decreases.

20.3 Under what circumstances is an EmptyStackException thrown?

20.4 Fill in the blanks in each of the following statements:

a) Bits in the result of an expression using operator _________ are set to 1 if the corresponding bits in each operand are set to 1. Otherwise, the bits are set to zero.

b) Bits in the result of an expression using operator _________ are set to 1 if at least one of the corresponding bits in either operand is set to 1. Otherwise, the bits are set to zero.

c) Bits in the result of an expression using operator _________ are set to 1 if exactly one of the corresponding bits in either operand is set to 1. Otherwise, the bits are set to zero.

d) The bitwise AND operator (\&) is often used to _________ bits, that is, to select certain bits from a bit string while zeroing others.

e) The _________ operator is used to shift the bits of a value to the left.

f) The _________ operator shifts the bits of a value to the right with sign extension, and the _________ operator shifts the bits of a value to the right with zero extension.

ANSWERS TO SELF-REVIEW EXERCISES

20.1 a) Vector. b) double. c) trimToSize.

20.2 a) False; a Vector stores only Objects. A program must use the type-wrapper classes (Byte, Short, Integer, Long, Float, Double, Boolean and Character) from package java.lang to create Objects containing the primitive data types. b) False; as the load fac-
tor increases, there are fewer available slots relative to the total number of slots, so the chance of selecting an occupied slot (a collision) with a hashing operation increases.

20.3 When a program calls pop or peek on an empty Stack object, an EmptyStackException occurs.

20.4 a) bitwise AND (&). b) bitwise inclusive OR (|). c) bitwise exclusive OR (^). d) mask. e) left shift operator (<<). f) right shift operator with sign extension (>>), right shift operator with zero extension (>>>).

EXERCISES

20.5 Define each of the following terms in the context of hashing:
   a) key
   b) collision
   c) hashing transformation
   d) load factor
   e) space–time trade-off
   f) Hashtable class
   g) capacity of a Hashtable

20.6 Explain briefly the operation of each of the following methods of class Vector:
   a) addElement
   b) insertElementAt
   c) setElementAt
   d) removeElement
   e) removeAllElements
   f) removeElementAt
   g) firstElement
   h) lastElement
   i) isEmpty
   j) contains
   k) indexOf
   l) trimToSize
   m) size
   n) capacity

20.7 Explain why inserting additional elements into a Vector object whose current size is less than its capacity is a relatively fast operation and why inserting additional elements into a Vector object whose current size is at capacity is a relatively slow operation.

20.8 In the text, we state that the default capacity increment of doubling the size of a Vector might seem wasteful of storage, but it is actually an efficient way for Vectors to grow quickly to be “about the right size.” Explain this statement. Explain the pros and cons of this doubling algorithm. What can a program do when it determines that the doubling is wasting space?

20.9 Explain the use of the Enumeration interface with objects of class Vector.

20.10 By extending class Vector, Java’s designers were able to create class Stack quickly. What are the negative aspects of this use of inheritance, particularly for class Stack?

20.11 Explain briefly the operation of each of the following methods of class Hashtable:
   a) put
   b) get
   c) remove
   d) isEmpty
20.12 Explain how to use the Random class to create pseudorandom numbers with the repeatability required for debugging purposes.

20.13 Use a Hashtable to create a reusable class for choosing one of the 13 predefined colors in class Color. The name of the color should be used as keys and the predefined Color objects should be used as values. Place this class in a package that can be imported into any Java program. Use your new class in an application that allows the user to select a color and draw a shape in that color.

20.14 Modify your solution to Exercise 13.18—the polymorphic painting program—to store every shape the user draws in a Vector of MyShape objects. For the purpose of this exercise, create your own Vector subclass called ShapeVector that manipulates only MyShape objects. Provide the following capabilities in your program:
   a) Allow the user of the program to remove any number of shapes from the Vector by clicking an Undo button.
   b) Allow the user to select any shape on the screen and move it to a new location. This requires the addition of a new method to the MyShape hierarchy. The method’s first line should be

   ```java
   public boolean isInside()
   ```

   This method should be overridden for each subclass of MyShape to determine whether the coordinates where the user pressed the mouse button are inside the shape.
   c) Allow the user to select any shape on the screen and change its color.
   d) Allow the user to select any shape on the screen that can be filled or unfilled and change its fill state.

20.15 What does it mean when we state that a Properties object is a “persistent” Hashtable object? Explain the operation of each of the following methods of the Properties class:
   a) load
   b) store
   c) getProperty
   d) propertyNames
   e) list

20.16 Why might you want to use objects of class BitSet? Explain the operation of each of the following methods of class BitSet:
   a) set
   b) clear
   c) get
   d) and
   e) or
   f) xor
   g) size
   h) equals
   i) clone
   j) toString
   k) hashCode

20.17 Write a program that right shifts an integer variable 4 bits with sign extension and then right shifts the same integer variable 4 bits with zero extension. The program should print the integer in
bits before and after each shift operation. Run your program once with a positive integer and once with a negative integer.

20.18 Show how shifting an integer left by 1 can be used to simulate multiplication by 2 and how shifting an integer right by 2 can be used to simulate division by 2. Be careful to consider issues related to the sign of an integer.

20.19 Write a program that reverses the order of the bits in an integer value. The program should input the value from the user and call method `reverseBits` to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly. You might want to implement both a recursive and an iterative solution.

20.20 Modify your solution to Exercise 19.10 to use class `Stack`.

20.21 Modify your solution to Exercise 19.12 to use class `Stack`.

20.22 Modify your solution to Exercise 19.13 to use class `Stack`.
Objectives

• To understand what collections are.
• To understand Java 2’s new array capabilities.
• To use the collections framework implementations.
• To be able to use collections framework algorithms to manipulate various collections.
• To be able to use the collections framework interfaces to program polymorphically.
• To be able to use iterators to walk through the elements of a collection.
• To understand synchronization wrappers and modifiability wrappers.

I think this is the most extraordinary collection of talent, of human knowledge, that has ever been gathered together at the White House—with the possible exception of when Thomas Jefferson dines alone.
John F. Kennedy

The shapes a bright container can contain!
Theodore Roethke

Journey over all the universe in a map.
Miguel de Cervantes

It is an immutable law in business that words are words, explanations are explanations, promises are promises — but only performance is reality.
Harold S. Green
21.1 Introduction

In Chapter 19, we discussed how to create and manipulate data structures. The discussion was “low level,” in the sense that we painstakingly created each element of each data structure dynamically with `new` and modified the data structures by directly manipulating their elements and references to their elements. In this chapter, we consider the Java collections framework, which gives the programmer access to prepackaged data structures, as well as algorithms for manipulating those data structures.

With collections, instead of creating data structures, the programmer simply uses existing data structures, without concern for how the data structures are implemented. This methodology is a marvelous example of code reuse. Programmers can code faster and can expect excellent performance, maximizing execution speed and minimizing memory consumption. We will discuss the interfaces of the collections framework, the implementation classes, the algorithms that process them and the `iterators` that “walk” through them.

Some examples of collections are the cards you hold in a card game, your favorite songs stored in your computer and the real-estate records in your local registry of deeds (which map book numbers and page numbers to property owners). Java 2 provides an entire collections framework, whereas earlier versions of Java provided just a few collection classes, such as `Hashtable`, `Stack` and `Vector` (see Chapter 20), as well as built-in array capabilities. If you know C++, you will be familiar with its collections framework,

The Java collections framework provides ready-to-go, reusable componentry; you do not need to write your own collection classes. The collections are standardized so applications can share them easily, without having to be concerned with the details of their implementation. These collections are written for broad reuse. They are tuned for rapid execution as well as efficient use of memory. The collections framework encourages further reusability. As new data structures and algorithms are developed that fit this framework, a large base of programmers already will be familiar with the interfaces and algorithms implemented by those data structures.

### 21.2 Collections Overview

A collection is a data structure—actually, an object—that can hold other objects. The collection interfaces define the operations that a program can perform on each type of collection. The collection implementations execute the operations in particular ways, some more appropriate than others for specific kinds of applications. The collection implementations are carefully constructed for rapid execution and efficient use of memory. Collections encourage software reuse by providing convenient functionality.

The collections framework provides interfaces that define the operations to be performed generically on various types of collections. Some of the interfaces are `Collection`, `Set`, `List` and `Map`. Several implementations of these interfaces are provided within the framework. Programmers may also provide implementations specific to their own requirements.

The collections framework includes a number of other features that minimize the amount of coding programmers need to do to create and manipulate collections.

The classes and interfaces that comprise the collections framework are members of package `java.util`. In the next section, we begin our discussion by examining the capabilities that have been added for array manipulation.

### 21.3 Class Arrays

We begin our discussion of the collections framework by looking at class `Arrays`, which provides `static` methods for manipulating arrays. In Chapter 7, our discussion of array manipulation was “low level,” in the sense that we wrote the actual code to sort and search arrays. Class `Arrays` provides “high-level” methods, such as `binarySearch` for searching a sorted array, `equals` for comparing arrays, `fill` for placing values into an array and `sort` for sorting an array. These methods are overloaded for primitive-type arrays and `Object` arrays. Figure 21.1 demonstrates the use of these methods.

```java
// Fig. 21.1: UsingArrays.java
// Fig. 21.1: Using Java arrays.

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

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/12/01
public class UsingArrays {
    private int intValues[] = { 1, 2, 3, 4, 5, 6 };  
    private double doubleValues[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };  
    private int filledInt[], intValuesCopy[];
    
    // initialize arrays
    public UsingArrays() {
        filledInt = new int[ 10 ];   
        intValuesCopy = new int[ intValues.length ];
        Arrays.fill( filledInt, 7 );   // fill with 7s
        Arrays.sort( doubleValues );   // sort doubleValues
        System.arraycopy( intValues, 0, intValuesCopy, 0, intValues.length );
    }
    
    // output values in each array
    public void printArrays() {
        System.out.print( "doubleValues: " );
        for ( int count = 0; count < doubleValues.length; count++ )
            System.out.print( doubleValues[ count ] + " " );
        System.out.print( "\nintValues: " );
        for ( int count = 0; count < intValues.length; count++ )
            System.out.print( intValues[ count ] + " " );
        System.out.print( "\nfilledInt: " );
        for ( int count = 0; count < filledInt.length; count++ )
            System.out.print( filledInt[ count ] + " " );
        System.out.print( "\nintValuesCopy: " );
        for ( int count = 0; count < intValuesCopy.length; count++ )
            System.out.print( intValuesCopy[ count ] + " " );
        System.out.println();
    }
    
    // find value in array intValues
    public int searchForInt( int value ) {
        return Arrays.binarySearch( intValues, value );
    }
}
Line 18 calls `Arrays static` method `fill` to populate all 10 elements of array `filledInt` with 7s. Overloaded versions of `fill` allow the programmer to populate a specific range of elements with the same value.

Line 20 sorts the elements of array `doubleValues`. Overloaded versions of `sort` allow the programmer to sort a specific range of elements. `Arrays static` method `sort` orders the array’s elements in ascending order by default. We discuss how to sort in descending order later in the chapter.
Lines 22–23 copy `intValues` into array `intValuesCopy`. The first argument (`intValues`) passed to `System static` method `arraycopy` is the array from which elements are copied. The second argument (0) is the array (i.e., `intValues`) subscript that specifies the starting point in the range of elements to copy. This value can be any valid array subscript. The third argument (`intValuesCopy`) specifies the array that stores the copy. The fourth argument (0) specifies the subscript in the destination array (i.e., `intValuesCopy`) where the first copied element is stored. The last argument (`intValues.length`) specifies the number of source-array (i.e., `intValues`) elements to copy.

Line 55 calls `Arrays static` method `binarySearch` to perform a binary search on `intValues`, using `value` as the key. If `value` is found, `binarySearch` returns the subscript location where `value` was found. If `value` is not found, `binarySearch` returns a negative value. The negative value returned is based on the search key’s `insertion point`—the index where the key would be inserted in the binary search tree if this were an insert operation. After `binarySearch` determines the insertion point, it changes the insertion point’s sign to negative and subtracts 1 to obtain the return value. For example, in Fig. 21.1, the insertion point for the value 8763 is the element with subscript 6 in the array. Method `binarySearch` changes the insertion point to −6 and subtracts 1 from it, then returns the value −7. This return value is useful for adding elements to a sorted array.

**Common Programming Error 21.1**

Passing an unsorted array to `binarySearch` is a logic error. The value returned by `binarySearch` is undefined.

Lines 61 and 66 call method `equals` to determine whether the elements of two arrays are equivalent. If they are equal, method `equals` returns `true`; otherwise, it returns `false`.

One of the most important features of the collection framework is the ability to manipulate the elements of one collection type through a different collection type, regardless of the collection’s internal implementation. The `public` set of methods through which collections are manipulated is called a `view`.

Class `Arrays` provides `static` method `asList` for viewing an array as a `List` collection type (a type that encapsulates behavior similar to that of the linked lists created in Chapter 19; we will say more about `Lists` later in the chapter). A `List` view allows the programmer to manipulate the array programmatically as if it were a `List` by calling `List` methods. Any modifications made through the `List` view change the array, and any modifications made to the array change the `List` view. Figure 21.2 demonstrates method `asList`.

```
1 // Fig. 21.2: UsingAsList.java
2 // Using method asList
3
4 // Java core packages
5 import java.util.*;
6
7 public class UsingAsList {
8     private String values[] = { "red", "white", "blue" };
9     private List list;
10
```

**Fig. 21.2** Using `static` method `asList` (part 1 of 2).
Line 9 declares a List reference called list. Line 14 uses static Arrays method asList to obtain a fixed-size List view of array values.

Performance Tip 21.1

Arrays.asList creates a fixed-size List that operates faster than any of the provided List implementations.

Common Programming Error 21.2

A List created with Arrays.asList is fixed in size; calling methods add or remove throws an UnsupportedOperationException.

Line 15 calls List method set to change the contents of List element 1 to "green". Because the program views the array as a List, line 15 changes array element values[1] from "white" to "green". Any changes made to the List view are made to the underlying array object.
Software Engineering Observation 21.1

With the collections framework, there are many methods that apply to Lists and Collections that you would like to be able to use for arrays. Arrays.asList allows you to pass an array into a List or Collection parameter.

Line 23 calls List method size to get the number of items in the List. Line 24 calls List method get to retrieve an individual item from the List. Notice that the value returned by size is equal to the number of elements in array values and that the items returned by get are the elements of array values.

21.4 Interface Collection and Class Collections

Interface Collection is the root interface in the collections hierarchy from which interfaces Set (a collection that does not contain duplicates—discussed in Section 21.7) and List are derived. Interface Collection contains bulk operations (i.e., operations performed on the entire collection) for adding, clearing, comparing and retaining objects (also called elements) in the collection. Collections can also be converted to arrays. In addition, interface Collection provides a method that returns an Iterator. Iterators are similar to the Enumerations introduced in Chapter 20. The primary difference between an Iterator and an Enumeration is that Iterators can remove elements from a collection, whereas Enumerations cannot. Other methods of interface Collection enable a program to determine a collection’s size, a collection’s hash code and whether a collection is empty.

Software Engineering Observation 21.2

Collection is used commonly as a method parameter type to allow polymorphic processing of all objects that implement interface Collection.

Software Engineering Observation 21.3

Most collection implementations provide a constructor that takes a Collection argument, thereby allowing one collection type to be treated as another collection type.

Class Collections provides static methods that manipulate collections polymorphically. These methods implement algorithms for searching, sorting, and so on. You will learn more about these algorithms in Section 21.6. Other Collections methods include wrapper methods that return new collections. We discuss wrapper methods in Section 21.9 and Section 21.10.

21.5 Lists

A List is an ordered Collection that can contain duplicate elements. A List is sometimes called a sequence. Like arrays, Lists are zero based (i.e., the first element’s index is zero). In addition to the interface methods inherited from Collection, List provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching the elements and getting a ListIterator to access the elements.

Interface List is implemented by classes ArrayList, LinkedList and Vector. Class ArrayList is a resizable-array implementation of a List. Class ArrayList’s behavior and capabilities are similar to those of the Vector class, introduced in Chapter 20. A LinkedList is a linked-list implementation of a List.
Performance Tip 21.2

ArrayLists behave like unsynchronized Vectors and therefore execute faster than Vectors, because ArrayLists are not thread safe.

Software Engineering Observation 21.4

LinkedLists can be used to create stacks, queues, trees and deques (double-ended queues).

Figure 21.3 uses an ArrayList to demonstrate some of the capabilities of Collection interfaces. The program places Strings and Colors in an ArrayList and uses an Iterator to remove the Strings from the ArrayList collection.

```java
// Fig. 21.3: CollectionTest.java
// Using the Collection interface

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

public class CollectionTest {
    private String colors[] = { "red", "white", "blue" };
    // create ArrayList, add objects to it and manipulate it
    public CollectionTest() {
        ArrayList list = new ArrayList();
        // add objects to list
        list.add( Color.magenta );     // add a color object
        for ( int count = 0; count < colors.length; count++ )
            list.add( colors[ count ] );
        list.add( Color.cyan );        // add a color object
        // output list contents
        System.out.println( "ArrayList: " );
        for ( int count = 0; count < list.size(); count++ )
            System.out.print( list.get( count ) + " " );
        // remove all String objects
        removeStrings( list );
        // output list contents
        System.out.println( "\nArrayList after calling removeStrings: " );
        for ( int count = 0; count < list.size(); count++ )
            System.out.print( list.get( count ) + " " );
    }
}
```

Fig. 21.3 Using an ArrayList to demonstrate interface Collection (part 1 of 2).
Line 14 creates reference \texttt{list} and initializes it with an instance of an \texttt{ArrayList}. Lines 17–22 populate \texttt{list} with \texttt{Color} and \texttt{String} objects. Lines 25–28 output each element of \texttt{list}. Line 27 calls \texttt{List} method \texttt{size} to get the number of \texttt{ArrayList} elements. Line 28 uses \texttt{List} method \texttt{get} to retrieve individual element values. Line 31 calls programmer-defined method \texttt{removeStrings} (defined on lines 42–52), passing \texttt{list} to it as an argument. Method \texttt{removeStrings} deletes \texttt{Strings} from a collection. Lines 34–38 print the elements of \texttt{list} after \texttt{removeStrings} removes the \texttt{String} objects from the list. Notice that the output in Fig. 21.3 contains only \texttt{Colors}.

Method \texttt{removeStrings} declares one parameter of type \texttt{Collection} that allows any \texttt{Collection} to be passed as an argument to this method. The method accesses the elements of the \texttt{Collection} via an \texttt{Iterator}. Line 45 calls method \texttt{iterator} to get an \texttt{Iterator} for the \texttt{Collection}. The while loop condition on line 48 calls \texttt{Iterator} method \texttt{hasNext} to determine if the \texttt{Collection} contains any more elements. Method \texttt{hasNext} returns \texttt{true} if another element exists and \texttt{false} otherwise.

The if condition at line 50 calls \texttt{Iterator} method \texttt{next} to obtain a reference to the next element, then uses \texttt{instanceof} to determine whether the object is a \texttt{String}. If so, line 51 calls \texttt{Iterator} method \texttt{remove} to remove the \texttt{String} from the \texttt{Collection}. 

\begin{verbatim}
40 // remove String objects from Collection
41 public void removeStrings( Collection collection )
42 {
43     // get iterator
44     Iterator iterator = collection.iterator();
45     // loop while collection has items
46     while ( iterator.hasNext() )
47         if ( iterator.next() instanceof String )
48             iterator.remove();   // remove String object
49     }
50     // execute application
51     public static void main( String args[] )
52     {
53         new CollectionTest();
54     }
55 }
56 // end class CollectionTest
\end{verbatim}

\texttt{ArrayList: java.awt.Color[r=255,g=0,b=255] red white blue java.awt.Color[r=0,g=255,b=255]}

\texttt{ArrayList after calling removeStrings: java.awt.Color[r=255,g=0,b=255] java.awt.Color[r=0,g=255,b=255]}

\textbf{Fig. 21.3} Using an \texttt{ArrayList} to demonstrate interface \texttt{Collection} (part 2 of 2).
Common Programming Error 21.3

When iterating through a collection with an `Iterator`, use `Iterator` method `remove` to delete an element from the collection. Using the collection’s `remove` method will result in a `ConcurrentModificationException`.

Figure 21.4 demonstrates operations on `LinkedLists`. The program creates two `LinkedLists` that each contain `Strings`. The elements of one `List` are added to the other. Then, all the `Strings` are converted to uppercase, and a range of elements is deleted.

```java
// Fig. 21.4: ListTest.java
// Using LinkLists

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

public class ListTest {
    private String colors[] = { "black", "yellow", "green", "blue", "violet", "silver" };  
    private String colors2[] = { "gold", "white", "brown", "blue", "gray", "silver" };  

    // set up and manipulate LinkedList objects
    public ListTest() {
        LinkedList link = new LinkedList();
        LinkedList link2 = new LinkedList();

        // add elements to each list
        for ( int count = 0; count < colors.length; count++ ) {
            link.add( colors[ count ] );
            link2.add( colors2[ count ] );
        }

        link.addAll( link2 );           // concatenate lists
        link2 = null;                   // release resources

        printList( link );

        uppercaseStrings( link );

        printList( link );

        System.out.print( "Deleting elements 4 to 6..." );
        removeItems( link, 4, 7 );

        printList( link );
    }

    // output List contents
    public void printList( List list ) {
        System.out.println( "\nlist: " );
    }

    // Fig. 21.4 Using Lists and ListIterators (part 1 of 2).
```
Lines 16–17 create `LinkedLists link` and `link2`, respectively. Lines 20–23 call method `add` to append elements from arrays `colors` and `colors2` to the end of `LinkedLists link` and `link2`, respectively.

Line 25 calls method `addAll` to append all elements of `link2` to the end of `link`. Line 26 sets `link2` to `null`, so `LinkedList` can be garbage collected. Line 28 calls programmer-defined method `printList` (lines 41–49) to output the `link`’s contents.
Line 30 calls programmer-defined method `uppercaseStrings` (lines 52–63) to convert the `String` elements to uppercase; then line 32 calls `printList` to display the modified `Strings`. Line 34 calls programmer-defined method `removeItems` (lines 66–69) to remove the elements at positions 4 through 6 of the list.

Method `uppercaseStrings` (lines 52–63) changes lowercase `String` elements in the `List` passed to it to uppercase `Strings`. Line 54 calls method `listIterator` to get a bidirectional iterator (i.e., an iterator that can traverse a `List` backward or forward) for the `List`. The while condition calls method `hasNext` to determine whether the `List` contains another element. Line 57 gets the next `Object` from the `List` and assigns it to `object`. The if condition tests `object` to determine whether it is an `instanceof` class `String`. If so, lines 60–61 cast `object` to a `String`, call method `toUpperCase` to get an uppercase version of the `String` and call method `set` to replace the current `String` to which `iterator` refers with the `String` returned by method `toUpperCase`.

Programmer-defined method `removeItems` (lines 66–69) removes a range of items from the list. Line 68 calls method `subList` to obtain a portion of the `List` called a `sublist`. The sublist is simply a view into the `List` on which `subList` is called. Method `subList` takes two arguments—the beginning index for the sublist and the ending index for the sublist. Note that the ending index is not part of the range of the sublist. In this example, we pass 4 for the beginning index and 7 for the ending index to `subList`. The sublist returned is the elements with indices 4 through 6. Next, the program calls method `clear` on the sublist to remove the elements of the sublist from the `List`. Any changes made to a sublist are made to the original `List`.

Figure 21.5 uses method `toArray` to get an array from a collection (i.e., `LinkedList`). The program adds a series of `Strings` to a `LinkedList` and calls method `toArray` to get an array from the `LinkedList`.

```java
// Fig. 21.5: UsingToArray.java
// Using method toArray

public class UsingToArray {
  public UsingToArray()
  {
    LinkedList links;
    String colors[] = { "black", "blue", "yellow" };

    links = new LinkedList( Arrays.asList( colors ) );

    links.addLast( "red" ); // add as last item
    links.add( "pink" );    // add to the end
    links.add( 3, "green" ); // add at 3rd index
    links.addFirst( "cyan" ); // add as first item
  }
}
```

Fig. 21.5 Using method `toArray` (part 1 of 2).
Line 15 constructs a `LinkedList` containing the elements of array `colors` and assigns the `LinkedList` to `links`. Line 17 calls method `addLast` to add "red" to the end of `links`. Lines 18–19 call method `add` to add "pink" as the last element and "green" as the element at index 3 (i.e., the fourth element). Line 20 calls `addFirst` to add "cyan" as the new first item in the `LinkedList`. [Note: When "cyan" is added as the first element, "green" becomes the fifth element in the `LinkedList`.]

Lines 23–24 call method `toArray` to get a `String` array from `links`. The array is a copy of the list elements—modifying the contents of the array does not modify the `LinkedList`. The array passed to method `toArray` is of the same data type as that returned by `toArray`. If the number of elements in the array is greater than the number of elements in the `LinkedList`, `toArray` copies the list elements into its array argument and returns that array. If the `LinkedList` has more elements than the number of elements in the array passed to `toArray`, `toArray` allocates a new array of the same type it receives as an argument, copies the list elements into the new array and returns the new array.

**Common Programming Error 21.4**

Passing an array that contains data to `toArray` can create logic errors. If the number of elements in the array is smaller than the number of elements in the `Object` calling `toArray`, new memory is allocated to store the `Object`’s elements—without preserving the array’s elements. If the number of elements in the array is greater than the number of elements in the `Object`, the elements of the array (starting at subscript 0) are overwritten with the `Object`’s elements. Array elements that are not overwritten retain their values.

Fig. 21.5 Using method `toArray` (part 2 of 2).
21.6 Algorithms

The collections framework provides a variety of high-performance algorithms for manipulating collection elements. These algorithms are implemented as static methods. Algorithms sort, binarySearch, reverse, shuffle, fill and copy operate on Lists. Algorithms min and max operate on Collections.

Algorithm reverse reverses the elements of a List, fill sets every List element to refer to a specified Object and copy copies references from one List into another.

**Software Engineering Observation 21.5**
The collections framework algorithms are polymorphic. That is, each algorithm can operate on objects that offer given interfaces without concern to the underlying implementations.

21.6.1 Algorithm sort

Algorithm sort sorts the elements of a List. The order is determined by the natural order of the elements’ type. The sort call may specify as a second argument a Comparator object that specifies how to determine the ordering of the elements.

Algorithm sort uses a stable sort (i.e., a sort that does not reorder equivalent elements while sorting). The sort algorithm is fast. For readers who have studied some complexity theory in data structures or algorithms courses, this sort runs in $n \log(n)$ time. (Readers not familiar with complexity theory, may rest assured that this algorithm is extremely fast).

**Software Engineering Observation 21.6**
The Java API documentation sometimes provides implementation details. For example, sort is implemented as a modified merge sort. Avoid writing code that is dependent on implementation details, because they can change.

Figure 21.6 uses algorithm sort to order the elements of an ArrayList into ascending order (line 21).

```java
// Fig. 21.6: Sort1.java
// Using algorithm sort

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

public class Sort1 {
    private static String suits[] =
        { "Hearts", "Diamonds", "Clubs", "Spades" };

    // display array elements
    public void printElements()
    {
        // create ArrayList
        ArrayList list = new ArrayList( Arrays.asList( suits ) );

        // output list
        System.out.println( "Unsorted array elements:
" + list );
    }
}
```

Fig. 21.6 Using algorithm sort (part 1 of 2).
Figure 21.7 sorts the same Strings used in Fig. 21.6 into descending order. The example introduces the Comparator object, for sorting a Collection’s elements in a different order.

```java
// Fig. 21.7: Sort2.java
// Using a Comparator object with algorithm sort

public class Sort2 {
    private static String suits[] = {
        "Hearts", "Diamonds", "Clubs", "Spades" 
    };

    // output List elements
    public void printElements()
    {
        // create List
        List list = Arrays.asList( suits );

        // sort in descending order using a comparator
        Collections.sort( list, Collections.reverseOrder() );
    }
}
```

Fig. 21.7 Using a Comparator object in sort (part 1 of 2).
Line 21 calls `Collections`’s method `sort` to order the `List` view of the array into descending order. The `static Collections` method `reverseOrder` returns a `Comparator` object that represents the collection’s reverse order. For sorting a `List` view of a `String` array, the reverse order is a *lexicographical comparison*—the comparator compares the Unicode values that represent each element—in descending order.

### 21.6.2 Algorithm `shuffle`

Algorithm `shuffle` randomly orders a `List`’s elements. In Chapter 10, we presented a card shuffling and dealing simulation where we used a loop to shuffle a deck of cards. In Fig. 21.8, we use algorithm `shuffle` to shuffle the deck of cards. Much of the code is the same as in Fig. 10.19. The shuffling of the deck occurs on line 63, which calls `static Collections` method `shuffle` to shuffle the array through the array’s `List` view.

```java
// Fig. 21.8: Cards.java
// Using algorithm shuffle

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

// class to represent a Card in a deck of cards
class Card {
    private String face;
    private String suit;

    // initialize a Card
    public Card( String initialface, String initialSuit )
    {
        face = initialface;
        suit = initialSuit;
    }
```
// return face of Card
public String getFace()
{
    return face;
}

// return suit of Card
public String getSuit()
{
    return suit;
}

// return String representation of Card
public String toString()
{
    StringBuffer buffer =
        new StringBuffer( face + " of " + suit );
    buffer.setLength( 20 );
    return buffer.toString();
}

} // end class Card

// class Cards definition
public class Cards {
    private static String suits[] =
    { "Hearts", "Clubs", "Diamonds", "Spades" };
    private static String faces[] = { "Ace", "Deuce", "Three",
                                     "Jack", "Queen", "King" };
    private List list;

    // set up deck of Cards and shuffle
    public Cards()
    {
        Card deck[] = new Card[ 52 ];

        for ( int count = 0; count < deck.length; count++ )
            deck[ count ] = new Card( faces[ count % 13 ],
                                     suits[ count / 13 ] );

        list = Arrays.asList( deck );   // get List
        Collections.shuffle( list );    // shuffle deck
    }

    // output deck
    public void printCards()
    {
        int half = list.size() / 2 - 1;
21.6.3 Algorithms reverse, fill, copy, max and min

Class Collections provides algorithms for reversing, filling and copying Lists. Algorithm reverse reverses the order of the elements in a List, and algorithm fill overwrites elements in a List with a specified value. The fill operation is useful for reinitializing a List. Algorithm copy takes two arguments: A destination List and a source List. Each source List element is copied to the destination List. The destination List must be at least as long as the source List: otherwise, an IndexOutOfBoundsException is thrown. If the destination List is longer, the elements not overwritten are unchanged.
Each of the algorithms we have seen so far operates on Lists. Algorithms `min` and `max` each operate on Collections.

Algorithm `min` returns the smallest element in a List (remember a List is a Collection) and algorithm `max` returns the largest element in a List. Both of these algorithms can be called with a Comparator object as a second argument. Figure 21.9 demonstrates the use of algorithms `reverse`, `fill`, `copy`, `min` and `max`.

```java
// Fig. 21.9: Algorithms1.java
// Using algorithms reverse, fill, copy, min and max

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

public class Algorithms1 {
    private String letters[] = { "P", "C", "M" }, lettersCopy[];
    private List list, copyList;

    // create a List and manipulate it with algorithms from
    // class Collections
    public Algorithms1() {
        list = Arrays.asList( letters );     // get List
        lettersCopy = new String[3];
        copyList = Arrays.asList( lettersCopy );

        System.out.println( "Printing initial statistics: " );
        printStatistics( list );

        Collections.reverse( list );         // reverse order
        System.out.println( "\nPrinting statistics after " +
                "calling reverse: " );
        printStatistics( list );

        Collections.copy( copyList, list );  // copy List
        System.out.println( "\nPrinting statistics after " +
                "copying: " );
        printStatistics( copyList );

        System.out.println( "\nPrinting statistics after " +
                "calling fill: " );
        Collections.fill( list, "R" );
        printStatistics( list );
    }

    // output List information
    private void printStatistics( List listRef ) {
        System.out.print( "The list is: " );
        for ( int k = 0; k < listRef.size(); k++ )
            System.out.print( listRef.get( k ) + " " );
    }
}
```

Fig. 21.9 Using algorithms `reverse`, `fill`, `copy`, `max` and `min` (part 1 of 2).
21.6.4 Algorithm binarySearch

Earlier in this text, we studied the high-speed binary search algorithm. This algorithm is built right into the Java collections framework. The binarySearch algorithm locates an Object in a List (i.e., LinkedList, Vector or ArrayList) If the Object is found, the index (position relative to 0) of that Object is returned. If the Object is not found, binarySearch returns a negative value. Algorithm binarySearch determines this negative value by first calculating the insertion point and changing the insertion point’s sign to negative. Finally, binarySearch subtracts one from the insertion point to obtain the return value.
Figure 21.10 uses the `binarySearch` algorithm to search for a series of `Strings` in an `ArrayList`.

```java
// Fig. 21.10: BinarySearchTest.java
// Using algorithm binarySearch

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

public class BinarySearchTest {
    private String colors[] = { "red", "white", "blue", "black", "yellow", "purple", "tan", "pink" };    
    private ArrayList list;        // ArrayList reference

    // create, sort and output list
    public BinarySearchTest() {
        list = new ArrayList( Arrays.asList( colors ) );
        Collections.sort( list );   // sort the ArrayList
        System.out.println( "Sorted ArrayList: " + list );
    }

    // search list for various values
    public void printSearchResults() {
        printSearchResultsHelper( colors[3] ); // first item
        printSearchResultsHelper( colors[0] ); // middle item
        printSearchResultsHelper( colors[7] ); // last item
        printSearchResultsHelper( "aardvark" ); // below lowest
        printSearchResultsHelper( "goat" );      // does not exist
        printSearchResultsHelper( "zebra" );     // does not exist
    }

    // helper method to perform searches
    private void printSearchResultsHelper( String key ) {
        int result = 0;
        System.out.println( "\nSearching for: "+ key );
        result = Collections.binarySearch( list, key );
        System.out.println( ( result >= 0 ? "Found at index " + result : "Not Found (" + result + ")") );
    }

    // execute application
    public static void main( String args[] ) {
        new BinarySearchTest().printSearchResults();
    }
}
```

Fig. 21.10 Using algorithm `binarySearch` (part 1 of 2).
Line 16 calls Collections method sort to sort list into ascending order. Line 37 calls Collections method binarySearch to search list for the specified key. Method binarySearch takes a List as the first argument and an Object as the second argument. An overloaded version of binarySearch takes a Comparator object as its third argument, to specify how binarySearch should compare elements.

If the search key is found, method binarySearch returns the List index of the element containing the search key. When a search key is found in the List, the value returned by binarySearch is greater than or equal to zero. If the search key is not found, method binarySearch returns a negative number.

Software Engineering Observation 21.7
Java does not guarantee which item will be found first when a binarySearch is performed on a List containing multiple elements equivalent to the search key.

21.7 Sets

A Set is a Collection that contains unique elements (i.e., no duplicate elements). The collections framework contains two Set implementations: —HashSet and TreeSet. HashSet stores its elements in a hash table, and TreeSet stores its elements in a tree. Figure 21.11 uses a HashSet to remove duplicate Strings from an ArrayList.

Programmer-defined method printNonDuplicates (lines 23–35) takes a Collection argument. Line 26 constructs a HashSet from the Collection received as an argument to printNonDuplicates. When the HashSet is constructed, it removes any duplicates in the Collection. By definition, Sets do not contain any duplicates. Line 27 gets an Iterator for the HashSet. The while loop (lines 31–32) calls Iterator methods hasNext and next to access the HashSet elements.

Interface SortedSet extends Set and maintains its elements in sorted order (i.e., the elements’ natural order or an order specified by a Comparator). Class TreeSet implements SortedSet.
// Fig. 21.11: SetTest.java
// Using a HashSet to remove duplicates

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

public class SetTest {
    private String colors[] = { "red", "white", "blue", "green", "gray", "orange", "tan", "white", "cyan", "peach", "gray", "orange" };

    // create and output ArrayList
    public SetTest()
    {
        ArrayList list;

        list = new ArrayList( Arrays.asList( colors ) );
        System.out.println( "ArrayList: " + list );
        printNonDuplicates( list );
    }

    // create set from array to eliminate duplicates
    public void printNonDuplicates( Collection collection )
    {
        // create a HashSet and obtain its iterator
        HashSet set = new HashSet( collection );
        Iterator iterator = set.iterator();

        System.out.println( "Nonduplicates are: " );

        while ( iterator.hasNext() )
        {
            System.out.print( iterator.next() + " " );
        }
        System.out.println();
    }

    // execute application
    public static void main( String args[] )
    {
        new SetTest();
    }
} // end class SetTest

ArrayList: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]
Nonduplicates are:
orange cyan green tan white blue peach red gray

Fig. 21.11 Using a HashSet to remove duplicates.
The program of Fig. 21.12 places Strings into a TreeSet. The Strings are sorted automatically when they are added to the TreeSet. Also, range-view methods (i.e., methods that enable a program to view a portion of a collection) are demonstrated in this example.

```java
// Fig. 21.12: SortedSetTest.java
// Using TreeSet and SortedSet

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

public class SortedSetTest {
    private static String names[] = { "yellow", "green", "black", "tan", "grey", "white", "orange", "red", "green" };

    // create a sorted set with TreeSet, then manipulate it
    public SortedSetTest()
    {
        TreeSet tree = new TreeSet( Arrays.asList( names ) );

        System.out.println( "set: " );
        printSet( tree );

        // get headSet based upon "orange"
        System.out.print( "headSet ("orange") : " );
        printSet( tree.headSet( "orange" ) );

        // get tailSet based upon "orange"
        System.out.print( "tailSet ("orange") : " );
        printSet( tree.tailSet( "orange" ) );

        // get first and last elements
        System.out.println( "first: " + tree.first() );
        System.out.println( "last : " + tree.last() );
    }

    // output set
    public void printSet( SortedSet set )
    {
        Iterator iterator = set.iterator();
        while ( iterator.hasNext() )
            System.out.print( iterator.next() + " " );
        System.out.println();
    }

    // execute application
    public static void main( String args[] )
    {
        new SortedSetTest();
    }
}
```

Fig. 21.12 Using SortedSets and TreeSet.
Line 14 constructs a TreeSet object containing the elements of names and assigns a reference to this object to tree. Line 21 calls method headSet to get a subset of the TreeSet less than "orange". Any changes made to the subset are made to the TreeSet (i.e., the subset returned is a view of the TreeSet). Line 25 calls method tailSet to get a subset greater than or equal to "orange". Like headSet, any changes made through the tailSet view are made to the TreeSet. Lines 28–29 call methods first and last to get the smallest and largest elements, respectively.

Programmer-defined method printSet (lines 33–41) takes a SortedSet (e.g., a TreeSet) as an argument and prints it. Line 35 gets an Iterator for the Set. The body of the while loop prints each element of the SortedSet.

### 21.8 Maps

Maps associate keys to values and cannot contain duplicate keys (i.e., each key can map to only one value; this type of mapping is called one-to-one mapping). Maps differ from Sets in that Maps contain keys and values, whereas Sets contain only keys. Classes HashMap and TreeMap implement the Map interface. HashMaps store elements in HashTables, and TreeMaps store elements in trees. Interface SortedMap extends Map and maintains its keys in sorted order (i.e., the elements’ natural order or an order, specified by a Comparator). Class TreeMap implements SortedMap.

Figure 21.13 uses a HashMap to count the number of Strings that begin with a given letter. [Note: Unlike class Hashtable, class HashMap allows a null key and null values].

```java
// Fig. 21.13: MapTest.java
// Using a HashMap to store the number of words that begin with a given letter
// Java core packages
import java.util.*;

public class MapTest {
    private static String names[] = { "one", "two", "three", "four", "five", "six", "seven", "two", "ten", "four" };
}```
Line 15 constructs `HashMap map`. The `for` loop on lines 18–32 uses `map` to store the number of words in array `names` that begin with a given letter. Lines 19–20 call method `get` to retrieve a `Character` (the first letter of a `String` in `names`) from the
HashMap. If the HashMap does not contain a mapping for the Character, get returns null. If the HashMap does contain the mapping for the Character, its mapping value is returned as an Object. The returned value is cast to Integer and assigned to i.

If i is null, the Character is not in the HashMap, and lines 25–27 call method put to write an Integer containing 1 to the HashMap. The Integer value stored in the HashMap is the number of words beginning with that Character.

If the Character is in the HashMap, lines 29–31 increment the Integer counter by one and write the updated counter to the HashMap. A HashMap cannot contain duplicates, so put replaces the previous Integer object with the new one.

Programmer-defined method printMap takes one Map argument and prints it, using method toString. Lines 43–44 call methods size and isEmpty to get the number of values in the Map and a boolean indicating whether the Map is empty, respectively.

21.9 Synchronization Wrappers

In Chapter 15, we discussed multithreading. The built-in collections are unsynchronized. Concurrent access to a Collection by multiple threads could cause indeterminate results or fatal errors. To prevent potential threading problems, synchronization wrappers are used around collection classes that might be accessed by multiple threads. A wrapper class receives method calls, adds some functionality for thread safety and delegates the calls to the wrapped class.

The Collections API provides a set of public static methods for converting collections to synchronized versions. Method headers for the synchronization wrappers are listed in Fig. 21.14.

21.10 Unmodifiable Wrappers

The Collections API provides a set of public static methods for converting collections to unmodifiable versions (called unmodifiable wrappers) of those collections. Method headers for these methods are listed in Fig. 21.15. Unmodifiable wrappers throw UnsupportedOperationExceptions if attempts are made to modify the collection.

Software Engineering Observation 21.8

When creating an unmodifiable wrapper, not holding a reference to the backing collection ensures nonmodifiability.

public static method header

Collection synchronizedCollection( Collection c )
List synchronizedList( List aList )
Set synchronizedSet( Set s )
SortedSet synchronizedSortedSet( SortedSet s )
Map synchronizedMap( Map m )
SortedMap synchronizedSortedMap( SortedMap m )

Fig. 21.14 Synchronization wrapper methods.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/12/01
Software Engineering Observation 21.9

You can use an unmodifiable wrapper to create a collection that offers read-only access to others, while allowing read–write access to yourself. You do this simply by giving others a reference to the unmodifiable wrapper while you also retain a reference to the wrapped collection itself.

21.11 Abstract Implementations

The collections framework provides various abstract implementations (i.e., “bare bones” implementations of collection interfaces from which the programmer can quickly “flesh out” complete customized implementations). These abstract implementations are a thin Collection implementation called an AbstractCollection, a thin List implementation with random-access backing called an AbstractList, a thin Map implementation called an AbstractMap, a thin List implementation with sequential-access backing called an AbstractSequentialList and a thin Set implementation called an AbstractSet.

To write a custom implementation, begin by selecting as a base the abstract-implementation class that best meets your needs. Next, implement each of the class’s abstract methods. Then, if your collection is to be modifiable, override any concrete methods that prevent modification.

21.12 (Optional) Discovering Design Patterns: Design Patterns Used in Package java.util

In this section, we use the material on data structures and collections discussed in Chapters 19, 20 and 21 to identify classes from package java.util that use design patterns. This section concludes our treatment of design patterns.

21.12.1 Creational Design Patterns

We conclude the discussion of creational design patterns by discussing the Prototype design pattern.

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 the drawing program design of Exercise 9.28—class-
es **MyLine**, **MyOval** and **MyRect** represent “shape” classes that extend abstract superclass **MyShape**. We could modify this exercise to allow the user to create, copy and paste new instances of class **MyLine** into the program. The *Prototype design pattern* allows an object—called a *prototype*—to return a copy of that prototype to a requesting object—called a *client*. 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 class **java.lang.Object** and interface **java.lang.Cloneable**—any object from a class implementing **Cloneable** can use method *clone* to copy itself. Specifically, method *clone* creates a copy of an object, then returns a reference to that object. If we designate class **MyLine** as the prototype for Exercise 9.28, then class **MyLine** must implement interface **Cloneable**. To create a new line in our drawing, we clone the **MyLine** prototype. To copy a preexisting line, we clone that object. Method *clone* also is useful in methods that return a reference to an object, but the developer does not want that object to be altered through that reference—method *clone* returns a reference to the copy of the object instead of returning that object’s reference. For more information of interface **Cloneable**, visit

[www.java.sun.com/j2se/1.3/docs/api/java/lang/Cloneable.html](http://www.java.sun.com/j2se/1.3/docs/api/java/lang/Cloneable.html)

### 21.12.2 Behavioral Design Patterns

We conclude the discussion of behavioral design patterns by discussing the Iterator design pattern.

**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 any data structure without knowing the data structure’s behavior (such as traversing the structure or removing an element from that structure) or how that data structure stores objects. Instructions for traversing the data structure and accessing its elements are stored in a separate object called an *iterator*. Each data structure can create an iterator—each iterator implements methods of a common interface to traverse the data structure and access its data. An object can traverse two differently structured data structures—such as a linked list and a hash table—in the same manner, because both data structures contain an iterator object that belongs to a class implementing a common interface. Java provides interface **Iterator** from package **java.util**, which we discussed in Section 21.5—class **CollectionTest** (Fig 21.3) uses an **Iterator** object.

### 21.12.3 Conclusion

In our optional “Discovering Design Patterns” sections, we have introduced the importance, usefulness and prevalence of design patterns. We have mentioned that in their book *Design Patterns, Elements of Reusable Object-Oriented Software*, the “gang of four” described 23 design patterns that provide proven strategies for building systems. Each pattern belongs to one of three pattern categories: creational, which address issues related to object creation; structural, which provide ways to organize classes and objects in a system; and behavioral, which offer strategies to model how objects collaborate with one another in a system.
Of the 23 design patterns, we discussed 18 of the more popular ones used by the Java community. In Sections 9.24, 13.18, 15.13, 17.11 and 21.12, we divided the discussion according to how certain Java packages—such as package `java.awt`, `javax.swing`, `java.io`, `java.net` and `java.util`—use these design patterns. We also discussed patterns not described by the “gang of four,” such as concurrency patterns, which are useful in multithreaded systems, and architectural patterns, which help designers assign functionality to various subsystems in a system. We have motivated each pattern—that is, explained why that pattern is important and how it may be used. When appropriate, we supplied several examples in the form of real-world analogies (e.g., the adapter in the Adapter design pattern is similar to an adapter for a plug on an electrical device). We also gave examples of how Java packages take advantage of design patterns (e.g., Swing GUI components use the Observer design pattern to collaborate with their listeners to respond to user interactions). We also provided examples of how certain programs in *Java How to Program, Fourth edition* used design patterns (e.g., the elevator-simulation case study in our optional “Thinking About Objects” sections uses the State design pattern to represent a `Person` object’s location in the simulation).

We hope that you view our “Discovering Design Patterns” sections as a beginning to further study of design patterns. If you have not done so already, we recommend that you visit the many URLs we have provided in Section 9.24.5, Internet and World-Wide-Web Resources. We recommend that you then read the gang-of-four book. This information will help you build better systems using the collective wisdom of the object-technology industry.

If you have studied the optional sections in this book, you have been introduced to more substantial Java systems. If you have read our optional “Thinking About Objects” Sections, you have immersed yourself in a substantial design and Java implementation experience learning a disciplined approach to object-oriented design with the UML. If you have read our optional “Discovering Design Patterns” Sections, you have raised your awareness of the more advanced topic of design patterns.

We hope you continue your study of design patterns, and we would be most grateful if you would send your comments, criticisms and suggestions for improvement of *Java How to Program* to deitel@deitel.com. Good luck!

**SUMMARY**

- The Java collections framework gives the programmer access to prepackaged data structures, as well as algorithms for manipulating those data structures.
- Java 2 provides an entire collections framework, whereas earlier versions of Java provided just a few collection classes, like `HashTable` and `Vector`, as well as built-in array capabilities.
- A collection is a data structure; actually, it is an object that can hold other objects. The collection interfaces define the operations that can be performed on each type of collection.
- The collections framework includes a number of other features that minimize the amount of work programmers need to do to create and manipulate collections. This structure is an effective implementation of the notion of reuse.
- The classes and interfaces that compose the collections framework are members of the `java.util` package.
- Class `Arrays` provides **static** methods for manipulating arrays. Class `Arrays` methods include `binarySearch` for searching a sorted array, `equals` for comparing arrays, `fill` for placing items in an array, `sort` for sorting an array and `asList`. 
• Class **Arrays** provides method `asList` for getting a “**List**” view of the array. A **List** view allows the programmer to programmatically manipulate the array as if it were a **List**. This allows the programmer to treat an array as a collection. Any modifications made through the **List** view change the array, and any modifications to the array change the **List** view.

• Method `size` gets the number of items in a **List**, and method `get` gets an individual **List** element.

• Interface **Collection** is the root interface in the collections hierarchy from which interfaces **Set** and **List** are derived. Interface **Collection** contains bulk operations for adding, clearing, comparing and retaining objects in the collection.

• Interface **Collection** provides a method `iterator` for getting an **Iterator**.

• Class **Collections** provides static methods for manipulating collections. Many of the methods are implementations of polymorphic algorithms for searching, sorting and so on.

• A **List** is an ordered **Collection** that can contain duplicate elements. A **List** is sometimes called a sequence.

• Interface **List** is implemented by classes **ArrayList**, **LinkedList** and **Vector**. Class **ArrayList** is a resizable-array implementation of a **List**. **ArrayList** behavior and capabilities are similar to those of class **Vector**. A **LinkedList** is a linked-list implementation of a **List**.

• **Iterator** method `hasNext` determines whether a **Collection** contains another element. Method `hasNext` returns `true` if another element exists, `false` otherwise. Method `next` returns the next object in the **Collection** and advances the **Iterator**.

• Method `subList` gets a portion of a **List**, called a sublist. Any changes made to a sublist are also made to the **List** (i.e., the sublist is a “list view” of its corresponding **List** elements).

• Method `clear` removes elements from a **List**.

• Method `toArray` returns the contents of a collection as an array.

• Algorithms `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` and `copy` operate on **Lists**. Algorithms `min` and `max` operate on **Collections**. Algorithm `reverse` reverses the elements of a **List**. `fill` sets every **List** element to a specified **Object** and `copy` copies elements from one **List** into another **List**. Algorithm `sort` sorts the elements of a **List**.

• Algorithms `min` and `max` find the smallest item and the largest item in a **Collection**.

• The **Comparator** object provides a means of sorting a **Collection**’s elements in an order other than the **Collection**’s natural order.

• Method `reverseOrder` returns a **Comparator** object that represents the reverse order for a collection.

• Algorithm `shuffle` randomly orders the elements of a **List**.

• Algorithm `binarySearch` locates an **Object** in a **List**.

• A **Set** is a **Collection** that contains no duplicate elements. The collections framework contains two **Set** implementations: **HashSet** and **TreeSet**. **HashSet** stores its elements in a hash table; **TreeSet** stores its elements in a tree.

• Interface **SortedSet** extends **Set** and maintains its elements in sorted order. Class **TreeSet** implements **SortedSet**.

• Method `headSet` gets a subset of a **TreeSet** less than a specified element. Any changes made to the subset are made to the **TreeSet**. Method `tailSet` gets a subset greater than or equal to a specified element. Any changes made through the `tailSet` view are made to the **TreeSet**.

• **Maps** map keys to values and cannot contain duplicate keys. **Maps** differ from **Sets** in that **Maps** contain both keys and the values, whereas **Sets** contain only keys. Classes **HashMap** and
TreeMap implements the `Map` interface. HashMaps store elements in a `Hashtable`, and TreeMaps store elements in a tree.

- Interface `SortedMap` extends `Map` and maintains its elements in sorted order. Class `TreeMap` implements `SortedMap`.
- The built-in collections are unsynchronized. Concurrent access to a `Collection` by independent threads could cause indeterminate results. To prevent this, synchronization wrappers are used around classes that might be accessed by multiple threads.
- The `Collections` API provides a set of `public static` methods for converting collections to unmodifiable versions. Unmodifiable wrappers throw `UnsupportedOperationException` if attempts are made to modify the collection.
- The collections framework provides various abstract implementations (i.e., “bare bones” implementations of collection interfaces from which the programmer can quickly “flesh out” complete customized implementations).

**TERMINOLOGY**

- `AbstractCollection` class
- `AbstractList` class
- `AbstractMap` class
- `AbstractSequentialList` class
- `AbstractSet` class
- `add` method
- `addFirst` method
- `addLast` method
- algorithms
- `ArrayList` class
- arrays
- arrays as collections
- `Arrays.asList` method
- `bidirectional iterator`
- `binarySearch` algorithm
- `clear` method
- `Collection` interface
- collections
- `Collections` class
- collections framework
- collections placed in arrays
- `Comparator` object
- `copy` algorithm
- data structures
- delete an element from a collection
- `deque`
- double-ended queue (deque)
- duplicate elements
- `Enumeration` interface
- `fill` algorithm
- `HashMap` class
- `HashSet` class
- `Hashtable` class
- hashtable implementation
- `hasNext` method
- implementation classes
- insert an element into a collection interface
- `isEmpty` method
- iterator
- `Iterator` interface
- key
- lexicographical comparison
- `LinkedList` class
- `List` interface
- `ListIterator` map
- `Map` collection interface
- mapping keys to values
- mappings
- maps as collections
- `max` algorithm
- `min` algorithm
- modifiable collections
- natural ordering
- `next` method
- one-to-one mapping
- ordered collection
- ordering
- queue
- range-view methods
- `reverse` algorithm
- `reverseOrder` method
- sequence
- `Set` interface
- `shuffle` algorithm
- `size` method
- sort a `List`
SELF-REVIEW EXERCISES

21.1 Fill in the blanks in each of the following statements:
   a) Objects in a collection are called ________.
   b) An element in a List can be accessed by using the element’s ________.
   c) Lists are sometimes called ________.
   d) You can use a/an ________ to create a collection that offers only read-only access to
      others while allowing read–write access to yourself.
   e) ________ can be used to create stacks, queues, trees and deques (double-ended
      queues).

21.2 State whether each of the following is true or false. If false, explain why.
   a) A Set can contain duplicates.
   b) A Map can contain duplicate keys.
   c) A LinkedList can contain duplicates.
   d) Collections is an interface.
   e) Iterators can remove elements, while Enumerations cannot.

ANSWERS TO SELF-REVIEW EXERCISES

21.1 a) elements. b) index. c) sequences. d) unmodifiable wrapper. e) LinkedLists.

21.2 a) False. A Set cannot contain duplicate values.
   b) False. A Map cannot contain duplicate keys.
   c) True.
   d) False. Collections is a class, and Collection is an interface.
   e) True.

EXERCISES

21.3 Define each of the following terms:
   a) Collection
   b) Collections
   c) Comparator
   d) List

21.4 Briefly answer the following questions:
   a) What is the primary difference between a Set and a Map?
   b) Can a double-subscripted array be passed to Arrays method asList? If yes, how
      would an individual element be accessed?
   c) What must you do before adding a primitive data type (e.g., double) to a collection?

21.5 Explain briefly the operation of each of the following Iterator-related methods:
   a) iterator
   b) hasNext
   c) next
21.6 State whether each of the following is true or false. If false, explain why.
   a) Elements in a Collection must be sorted in ascending order before performing a binarySearch.
   b) Method first gets the first element in a TreeSet.
   c) A List created with Arrays.asList is resizable.
   d) Class Arrays provides static method sort for sorting array elements.

21.7 Rewrite method printList of Fig. 21.4 to use a ListIterator.

21.8 Rewrite lines 16–23 in Fig. 21.4 to be more concise by using the asList method and the LinkedList constructor that takes a Collection argument.

21.9 Write a program that reads in a series of first names and stores them in a LinkedList. Do not store duplicate names. Allow the user to search for a first name.

21.10 Modify the program of Fig. 21.13 to count the number of occurrences of all letters (e.g., five occurrences of "o" in the example). Display the results.

21.11 Write a program that determines and prints the number of duplicate words in a sentence. Treat uppercase and lowercase letters the same. Ignore punctuation.

21.12 Rewrite your solution to Exercise 19.8 to use a LinkedList collection.

21.13 Rewrite your solution to Exercise 19.9 to use a LinkedList collection.

21.14 Write a program that takes a whole-number input from a user and determines if it is prime. If the number is prime, add it to a JTextArea. If the number is not prime, display the prime factors of the number in a JLabel. Remember that a prime number’s factors are only 1 and the prime number itself. Every number that is not prime has a unique prime factorization. For example, consider the number 54. The factors of 54 are 2, 3, 3 and 3. When the values are multiplied together, the result is 54. For the number 54, the prime factors output should be 2 and 3. Use Sets as part of your solution.

21.15 Rewrite your solution to Exercise 22.21 to use a LinkedList.

21.16 Write a program that tokenizes (using class StreamTokenizer) a line of text input by the user and places each token in a tree. Print the elements of the sorted tree.
Objectives

- To understand the capabilities of the Java Media Framework (JMF).
- To understand the capabilities of the Java Sound API.
- To be able to play audio and video media with JMF.
- To be able to stream media over a network.
- To be able to capture, format and save media.
- To be able to play sounds with the Java Sound API.
- To be able to play, record, and synthesize MIDI with the Java Sound API.

TV gives everyone an image, but radio gives birth to a million images in a million brains.

Peggy Noonan

Noise proves nothing. Often a hen who has merely laid an egg cackles as if she had laid an asteroid.

Mark Twain, Following the Equator

A wide screen just makes a bad film twice as bad.

Samuel Goldwyn

Isn’t life a series of images that change as they repeat themselves?

Andy Warhol
22.1 Introduction

This chapter continues our multimedia discussions of Chapter 18 by introducing some of Java’s multimedia APIs that enable programmers to enhance applications with video and audio features. In recent years, the digital multimedia sector of the computer industry has experienced tremendous growth, as evidenced by the enormous quantity of multimedia content available on the Internet. Web sites have been transformed from text-based HTML pages to multimedia-intensive experiences. Advances in hardware and software technologies have allowed developers to integrate multimedia into the simplest applications. At the high end of multimedia applications, the video game industry has used multimedia programming to take advantage of the latest hardware technologies, such as 3D video cards that create virtual reality experiences for users.

Acknowledging that Java applications should support digital video and audio capabilities, Sun Microsystems, Intel and Silicon Graphics worked together to produce a multimedia API known as the **Java Media Framework (JMF)**. The JMF API is one of several multimedia APIs in Java. Using the JMF API, programmers can create Java applications that play, edit, stream and capture many popular media types. The first half of this chapter discusses the JMF API.

IBM and Sun developed the latest JMF specification—version 2.0. Sun provides a reference implementation—JMF 2.1.1—of the JMF specification which supports media file types such as **Microsoft Audio/Video Interleave (.avi)**, **Macromedia Flash 2 movies (.swf)**, **Future Splash (.spl)**, **MPEG Layer 3 Audio (.mp3)**, **Musical Instrument Digital Interface (MIDI; .mid)**, **MPEG-1 videos (.mpeg, .mpg)**, **QuickTime (.mov)**, **Sun Audio (.au)**, **Wave audio (.wav)**, **AIFF (.aiff)** and **GSM (.gsm)** files. The JMF also supports media from capture devices such as microphones and digital cameras.
In addition to the sample media clips provided with this chapter’s examples on the CD, many Web sites offer an abundant supply of free-for-download audio and video clips. You can download media clips from these sites (and many others on the Internet) and use them to test the examples in this chapter. We present a list of sites here to get you started. Free Audio Clips (www.freeaudioclips.com) is an excellent site for various types of audio files. The 13 Even site (www.13-even.com/media.html) provides audio and video clips in many formats for your personal use. If you are looking for MIDI audio files for use in Section 22.7, check out the free MIDI clips at www.freestuffgalore.commidi.asp. Funny Video Clips (www.video-clips.co.uk) offers entertaining material. Microsoft’s downloads site (msdn.microsoft.com/downloads) contains a multimedia section providing audio clips and other media.

Currently, JMF is available as an extension package separate from the Java 2 Software Development Kit. The CD that accompanies this book contains JMF 2.1.1. The most recent JMF implementation can be downloaded from the official JMF Web site:

java.sun.com/products/java-media/jmf

The JMF Web site provides versions of the JMF that take advantage of the performance features of the platform on which the JMF is running. For example, the JMF Windows Performance Pack provides extensive media and device support for Java programs running on Microsoft Windows platforms (Windows 95/98/NT 4.0/2000). JMF’s official Web site also provides continually updated support, information and resources for JMF programmers.

Portability Tip 22.1

Writing programs using JMF’s Windows Performance Pack reduces the portability of those programs to other operating systems.

The rest of this chapter discusses the Java Sound API and its extensive sound-processing capabilities. Internally, the JMF uses Java Sound for its audio functions. In Sections 22.5 through 22.7, we will demonstrate sampled audio playback and MIDI functionalities using Java Sound, a standard extension of the Java 2 Software Development Kit.

22.2 Playing Media

The JMF is commonly used to playback media clips in Java applications. Many applications such as financial managers, encyclopedias and games use multimedia to illustrate application features, present educational content and entertain users.

The JMF offers several mechanisms for playing media, the simplest of which is via objects that implement interface Player. Interface Player (package javax.media) extends Controller, which is a handler for JMF-supported media.

The following steps are needed to play a media clip:

1. Specify the media source.
2. Create a Player for the media.
3. Obtain the output media and Player controls.
4. Display the media and controls.

Class SimplePlayer (Fig. 22.1) is a simple Java media player program that demonstrates several common features of popular media players. The SimplePlayer demo
can play most JMF-supported media files with the possible exception of the latest versions of the formats. This application permits users to access files on the local computer that contain supported media types by clicking the **Open File** button. Clicking the **Open Location** button and specifying a media URL allows the user to access media from a media source, such as a *capture device*, a Web server, or a *streaming source*. A capture device (discussed in Section 22.3) reads media from audio and video devices such as microphones, CD players and cameras. A *Real-Time Transport Protocol (RTP) stream* is a stream of bytes sent over a network from a streaming server. An application buffers and plays the streaming media on the client computer.

```java
// Fig. 22.1: SimplePlayer.java
// Opens and plays a media file from local computer, public URL, or an RTP session

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

public class SimplePlayer extends JFrame {

    // Java media player
    private Player player;

    // visual content component
    private Component visualMedia;

    // controls component for media
    private Component mediaControl;

    // main container
    private Container container;

    // media file and media locations
    private File mediaFile;
    private URL fileURL;

    // constructor for SimplePlayer
    public SimplePlayer()
    {
        super( "Simple Java Media Player" );
        container = getContentPane();
        // panel containing buttons
        JPanel buttonPanel = new JPanel();
    }
}
```

Fig. 22.1 Playing media with interface **Player** (part 1 of 8).
container.add( buttonPanel, BorderLayout.NORTH );

// opening file from directory button
JButton openFile = new JButton( "Open File" );
buttonPanel.add( openFile );

// register an ActionListener for openFile events
openFile.addActionListener(
    new ActionListener() {
        // anonymous inner class to handle openFile events
        public void actionPerformed( ActionEvent event ) {
            mediaFile = getFile();

            if ( mediaFile != null ) {
                // obtain URL from file
                try {
                    fileURL = mediaFile.toURL();
                }

                // file path unresolvable
                catch ( MalformedURLException badURL ) {
                    badURL.printStackTrace();
                    showErrorMessage( "Bad URL" );
                }

                makePlayer( fileURL.toString() );
            }
        }
    }); // end call to method addActionListener

// URL opening button
JButton openURL = new JButton( "Open Locator" );
buttonPanel.add( openURL );

// register an ActionListener for openURL events
openURL.addActionListener(
    new ActionListener() {
        // anonymous inner class to handle openURL events
        public void actionPerformed( ActionEvent event ) {

            [Fig. 22.1 Playing media with interface Player (part 2 of 8).]
String addressName = getMediaLocation();

if ( addressName != null )
    makePlayer( addressName );

} // end ActionListener

// turn on lightweight rendering on players to enable
// better compatibility with lightweight GUI components
Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,
    Boolean.TRUE );

} // end SimplePlayer constructor

// utility method for pop-up error messages
public void showErrorMessage( String error )
{
    JOptionPane.showMessageDialog( this, error, "Error",
        JOptionPane.ERROR_MESSAGE );
}

// get file from computer
public File getFile()
{
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode(
        JFileChooser.FILES_ONLY );
    int result = fileChooser.showOpenDialog( this );
    if ( result == JFileChooser.CANCEL_OPTION )
        return null;
    else
        return fileChooser.getSelectedFile();
}

// get media location from user input
public String getMediaLocation()
{
    String input = JOptionPane.showInputDialog(
        this, "Enter URL" );
    if ( input != null && input.length() == 0 )
        return null;
    else
        return input;
}

Fig. 22.1 Playing media with interface Player (part 3 of 8).
// create player using media's location
public void makePlayer( String mediaLocation )
{
    // reset player and window if previous player exists
    if ( player != null )
    removePlayerComponents();

    // location of media source
    MediaLocator mediaLocator =
    new MediaLocator( mediaLocation );

    if ( mediaLocator == null ) {
        showErrorMessage( "Error opening file" );
        return;
    }

    // create a player from MediaLocator
    try {
        player = Manager.createPlayer( mediaLocator );

        // register ControllerListener to handle Player events
        player.addControllerListener(  
            new PlayerEventHandler() );

        // call realize to enable rendering of player's media
        player.realize();
    }

    // no player exists or format is unsupported
    catch ( NoPlayerException noPlayerException ) {
        noPlayerException.printStackTrace();
    }

    // file input error
    catch ( IOException ioException ) {
        ioException.printStackTrace();
    }

}  // end makePlayer method

// return player to system resources and
// reset media and controls
public void removePlayerComponents()
{
    // remove previous video component if there is one
    if ( visualMedia != null )
    container.remove( visualMedia );

    // remove previous media control if there is one
    if ( mediaControl != null )
    container.remove( mediaControl );

Fig. 22.1 Playing media with interface Player (part 4 of 8).
// stop player and return allocated resources
player.close();

// obtain visual media and player controls
public void getMediaComponents()
{
    // get visual component from player
    visualMedia = player.getVisualComponent();

    // add visual component if present
    if ( visualMedia != null )
        container.add( visualMedia, BorderLayout.CENTER );

    // get player control GUI
    mediaControl = player.getControlPanelComponent();

    // add controls component if present
    if ( mediaControl != null )
        container.add( mediaControl, BorderLayout.SOUTH );

}  // end method getMediaComponents

// handler for player's ControllerEvents
private class PlayerEventHandler extends ControllerAdapter {

    // prefetch media feed once player is realized
    public void realizeComplete( RealizeCompleteEvent realizeDoneEvent )
    {
        player.prefetch();
    }

    // player can start showing media after prefetching
    public void prefetchComplete( PrefetchCompleteEvent prefetchDoneEvent )
    {
        getMediaComponents();

        // ensure valid layout of frame
        validate();

        // start playing media
        player.start();

    }  // end prefetchComplete method

    // if end of media, reset to beginning, stop play
    public void endOfMedia( EndOfMediaEvent mediaEndEvent )
    {
        player.setMediaTime( new Time( 0 ) );
        player.stop();
    }

}  // end class PlayerEventHandler

Fig. 22.1  Playing media with interface Player (part 5 of 8).
public static void main( String args[] )
{
    SimplePlayer testPlayer = new SimplePlayer();
    testPlayer.setSize( 300, 300 );
    testPlayer.setLocation( 300, 300 );
    testPlayer.setDefaultCloseOperation( EXIT_ON_CLOSE );
    testPlayer.setVisible( true );
} // end class SimplePlayer
Chapter 22  Java Media Framework and Java Sound (on CD)  1245

Fig. 22.1  Playing media with interface **Player** (part 7 of 8).
A media clip must be processed before it is played. To process a media clip the program must access a media source, create a Controller for that source and output the media. Prior to output, users may perform optional formatting such as changing an AVI video to a QuickTime video. Although JMF hides low-level media processing (e.g. checking for file compatibility) from the programmer, both programmers and users can configure how a Player presents media. Section 22.3 and Section 22.4 reveal that capturing and streaming media follow the same guidelines. Section 22.8 lists several Web sites that have JMF-supported media contents.

Figure 22.1 introduces some key objects for playing media. The JMF extension package javax.media—imported in line 13—contains interface Player and other classes and interfaces needed for events. Line 18 declares a Player object to play media clips. Lines 30–31 declare the location of these clips as File and URL references.

Lines 21 and 24 declare Component objects for the video display and for holding the controls. Component mediaControl enables users to play, pause and stop the media clip. Component visualMedia displays the video portion of a media clip (if the media clip is a video). The JMF provides lightweight video renderers that are compatible with lightweight Swing components (See Chapter 13). Lines 107–108 in SimplePlayer’s constructor specify that the Player should draw its GUI components and video portion (if there
is one) using lightweight renderers so that the media player will look like other GUIs with Swing components. By default, **Player**’s video components are heavyweight components, which may not display correctly when mixed with lightweight Swing GUI components.

Before playing the media, **SimplePlayer** displays an initial GUI consisting of two buttons, **Open File** and **Open Locator**, that enable users to specify the media location. The event handlers for these two buttons (lines 52–78 and lines 90–101) perform similar functions. Each button prompts users for a media resource such as an audio or video clip, then creates a **Player** for the specified media. When the user clicks **Open File**, line 57 calls method **getfile** (lines 120–134) to prompt users to select a media file from the local computer. Line 63 calls the **File** method **toURL** to obtain a **URL** representation of the selected file’s name and location. Line 72 calls **SimplePlayer** method **makePlayer** (lines 150–187) to create a **Player** for the user-selected media. When users click **Open Locator**, line 95 invokes method **getMediaLocation** (lines 137–147), prompting users to input a **String** giving the media location. Line 98 calls **SimplePlayer** method **makePlayer** to create a **Player** for the media at the specified location. **Method makePlayer** (lines 150–187) makes the necessary preparations to create a **Player** of media clips. The **String** argument indicates the media’s location. Lines 153–154 invoke **SimplePlayer** method **removePlayerComponents** (lines 191–203) to remove the previous **Player**’s visual component and GUI controls from the frame before creating a new **Player**. Line 202 invokes **Player** method **close** to stop all **player** activity and to release system resources held by the previous **Player**.

**Method makePlayer** requires a pointer to the source from which the media is retrieved, which is accomplished by instantiating a new **MediaLocator** for the value given by the **String** argument (lines 157–158). A **MediaLocator** specifies the location of a media source, much like a **URL** typically specifies the location of a Web page. A **MediaLocator** can access media from capture devices and RTP sessions as well as from file locations. The **MediaLocator** constructor requires the media’s location as a **String**, so all **URLs** must be converted to **Strings** as in line 72.

**Method makePlayer** instantiates a new **Player** with a call to **Manager** method **createPlayer**. Class **Manager** provides static methods that enable programs to access most **JMF** resources. Method **createPlayer** opens the specified media source and determines the appropriate **Player** for the media source. Method **createPlayer** throws a **NoPlayerException** if an appropriate **Player** cannot be found for the media clip. An **IOException** is thrown if there are problems connecting to the media source.

**ControllerListener**s listen for the **ControllerEvents** that **Players** generate to track the progress of a **Player** in the media-handling process. Lines 170–171 register an instance of inner class **PlayerEventHandler** (lines 225–255) to listen for certain events that **player** generates. Class **PlayerEventHandler** extends class **ControllerAdapter**, which provides empty implementations of methods from interface **ControllerListener**. Class **ControllerAdapter** facilitates implementing **ControllerListener** for classes that need to handle only a few **Controller-Event** types.

**Players** confirm their progress while the processing media based on their state transitions. Line 174 invokes **Player** method **realize** to confirm all resources necessary to play media are available. Method **realize** places the **Player** in the **Realizing** state to indicate that it is connecting to and interacting with its media sources. When a **Player**
completes realizing, it generates a **RealizeCompleteEvent**—a type of **ControllerEvent** that occurs when a **Player** completes its transition to state **Realized**. This state indicates that the **Player** has completed all preparations needed to start processing the media. The program invokes method **realizeComplete** (lines 228–232) when **Player** generates a **RealizeCompleteEvent**.

Most media players have a buffering feature, which stores a portion of downloaded media locally so that users do not have to wait for an entire clip to download before playing it, as reading media data can take a long time. By invoking **Player** method **prefetch**, line 231 transitions the **player** to the **Prefetching** state. When a **Player** prefetches a media clip, the **Player** obtains exclusive control over certain system resources needed to play the clip. The **Player** also begins buffering media data to reduce the delay before the media clip plays.

When the **Player** completes prefetching, it transitions to state **Prefetched** and is ready to play media. During this transition, the **Player** generates a **ControllerEvent** of type **PrefetchCompleteEvent** to indicate that it is ready to display media. The **Player** invokes **PlayerEventHandler** method **prefetchComplete** (lines 235–246), which displays the **Player**’s GUI in the frame. After obtaining the hardware resources, the program can get the media components it requires. Line 238 invokes method **getMediaComponents** (lines 206–222) to obtain the GUI’s controls and the media’s visual component (if the media is a video clip) and attach them to the application window’s content pane. **Player** method **getVisualComponent** (line 209) obtains the visual component of the video clip. Similarly, line 216 invokes **Player** method **getControlPanelComponent** to return the GUI’s controls. The GUI (Fig. 22.1) typically provides the following controls:

1. A positioning slider to jump to certain points in the media clip.
2. A pause button.
3. A volume button that provides volume control by right clicking and a mute function by left clicking.
4. A media properties button that provides detailed media information by right clicking and frame rate control by left clicking.

**Look-and-Feel Observation 22.1**

Invoking **Player** method **getVisualComponent** yields **null** for audio files, because there is no visual component to display.

**Look-and-Feel Observation 22.2**

Invoking **Player** method **getControlPanelComponent** yields different sets of GUI controls depending on the media type. For example, media content streamed directly from a live conference does not have a progress bar because the length of the media is not pre-determined.

After validating the new frame layout (line 241), line 244 invokes **Player** method **start** (line 239) to start playing the media clip.

**Software Engineering Observation 22.1**

If the **Player** has not prefetched or realized the media, invoking **Player** method **start** prefetches and realizes the media.
Starting the **Player** takes less time if the **Player** has already prefetched the media before invoking start.

When the media clip ends, the **Player** generates a **ControllerEvent** of type **EndOfMediaEvent**. Most media players “rewind” the media clip after reaching the end so users can see or hear it again from the beginning. Method **endOfMedia** (lines 249–253) handles the **EndOfMediaEvent** and resets the media clip to its beginning position by invoking **Player** method **setMediaTime** with a new **Time** (package javax.media) of 0 (line 251). Method **setMediaTime** sets the position of the media to a specific time location in the media, and is useful for “jumping” to a different part of the media. Line 252 invites **Player** method **stop**, which ends media processing and places the **Player** in state **Stopped**. Invoking method **start** on a **Stopped Player** that has not been closed resumes media playback.

Often, it is desirable to configure the media before presentation. In the next section, we discuss interface **Processor**, which has more configuration capabilities than interface **Player**. **Processor** enables a program to format media and to save it to a file.

### 22.3 Formatting and Saving Captured Media

The Java Media Framework supports playing and saving media from capture devices such as microphones and video cameras. This type of media is known as **captured media**. Capture devices convert analog media into digitized media. For example, a program that captures an analog voice from a microphone attached to a computer can create a digitized file from that recording.

The JMF can access video capture devices that use **Video for Windows** drivers. Also, JMF supports audio capture devices that use the Windows **Direct Sound Interface** or the **Java Sound Interface**. The Video for Windows driver provides interfaces that enable Windows applications to access and process media from video devices. Similarly, Direct Sound and Java Sound are interfaces that enable applications to access sound devices such as hardware sound cards. The Solaris Performance Pack provides support for Java Sound and Sun Video capture devices on the Solaris platform. For a complete list of devices supported by JMF, visit JMF’s official Web site.

The **SimplePlayer** application presented in Fig. 22.1 allowed users to play media from a capture device. A **locator string** specifies the location of a capture device that the **SimplePlayer** demo accesses. For example, to test the **SimplePlayer**’s capturing capabilities, plug a microphone into a sound card’s microphone input jack. Typing the locator string *javasound:/* in the **Open Location** input dialog specifies that media should be input from the Java Sound-enabled capture device. The locator string initializes the **MediaLocator** that the **Player** needs for the audio material captured from the microphone.

Although **SimplePlayer** provides access to capture devices, it does not format the media or save captured data. Figure 22.2 presents a program that accomplishes these two new tasks. Class **CapturePlayer** provides more control over media properties via class **DataSource** (package javax.media.protocol). Class **DataSource** provides the connection to the media source, then abstracts that connection to allow users to manipulate it. This program uses a **DataSource** to format the input and output media. The **DataSource** passes the formatted output media to a **Controller**, which will format it
Further so that it can be saved to a file. The Controller that handles media is a Processor, which extends interface Player. Finally, an object that implements interface DataSource saves the captured, formatted media. The Processor object handles the flow of data from the DataSource to the DataSink object.

JMF and Java Sound use media sources extensively, so programmers must understand the arrangement of data in the media. The header on a media source specifies the media format and other essential information needed to play the media. The media content usually consists of tracks of data, similar to tracks of music on a CD. Media sources may have one or more tracks that contain a variety of data. For example, a movie clip may contain one track for video, one track for audio, and a third track for closed-captioning data for the hearing-impaired.

```java
// Fig. 22.2: CapturePlayer.java
// Presents and saves captured media

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

// Java extension packages
import javax.swing.*;
import javax.swing.event.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;
import javax.media.control.*;
import javax.media.datasink.*;

public class CapturePlayer extends JFrame {

    // capture and save button
    private JButton captureButton;

    // component for save capture GUI
    private Component saveProgress;

    // formats of device's media, user-chosen format
    private Format formats[], selectedFormat;

    // controls of device's media formats
    private FormatControl formatControls[];

    // specification information of device
    private CaptureDeviceInfo deviceInfo;

    // vector containing all devices' information
    private Vector deviceList;

Fig. 22.2 Formatting and saving media from capture devices (part 1 of 9).
```
// input and output data sources
private DataSource inSource, outSource;

// file writer for captured media
private DataSink dataSink;

// processor to render and save captured media
private Processor processor;

// constructor for CapturePlayer
public CapturePlayer()
{
    super( "Capture Player" );

    // panel containing buttons
    JPanel buttonPanel = new JPanel();
    getContentPane().add( buttonPanel );

    // button for accessing and initializing capture devices
    captureButton = new JButton( "Capture and Save File" );
    buttonPanel.add( captureButton, BorderLayout.CENTER );

    // register an ActionListener for captureButton events
    captureButton.addActionListener( new CaptureHandler() );

    // turn on light rendering to enable compatibility
    // with lightweight GUI components
    Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,
        Boolean.TRUE );

    // register a WindowListener to frame events
    addWindowListener(
    
        // anonymous inner class to handle WindowEvents
        new WindowAdapter() {

            // dispose processor
            public void windowClosing( WindowEvent windowEvent )
            {
                if ( processor != null )
                    processor.close();
            }

        } // end WindowAdapter
    ); // end call to method addWindowListener

} // end constructor

// action handler class for setting up device
private class CaptureHandler implements ActionListener {

Fig. 22.2 Formatting and saving media from capture devices (part 2 of 9).
// initialize and configure capture device
public void actionPerformed( ActionEvent actionEvent )
{
    // put available devices' information into vector
    deviceList =
        CaptureDeviceManager.getDeviceList( null );

    // if no devices found, display error message
    if ( ( deviceList == null ) ||
        ( deviceList.size() == 0 ) ) {
        showErrorMessage( "No capture devices found!" );
        return;
    }

    // array of device names
    String deviceNames[] = new String[ deviceList.size() ];

    // store all device names into array of
    // string for display purposes
    for ( int i = 0; i < deviceList.size(); i++ )
    {
        deviceInfo =
            ( CaptureDeviceInfo ) deviceList.elementAt( i );

        deviceNames[ i ] = deviceInfo.getName();
    }

    // get vector index of selected device
    int selectDeviceIndex =
        getSelectedDeviceIndex( deviceNames );

    if ( selectDeviceIndex == -1 )
        return;

    // get device information of selected device
    deviceInfo = ( CaptureDeviceInfo )
        deviceList.elementAt( selectDeviceIndex );

    formats = deviceInfo.getFormats();

    // if previous capture device opened, disconnect it
    if ( inSource != null )
        inSource.disconnect();

    // obtain device and set its format
    try {
        // create data source from MediaLocator of device
        inSource = Manager.createDataSource(
            deviceInfo.getLocator() );
    }
}

Fig. 22.2 Formatting and saving media from capture devices (part 3 of 9).
// get format setting controls for device
formatControls = ( ( CaptureDevice )
inSource ).getFormatControls();

// get user's desired device format setting
selectedFormat = getSelectedFormat( formats );

if ( selectedFormat == null )
    return;

setDeviceFormat( selectedFormat );
captureSaveFile();

} // end try

// unable to find DataSource from MediaLocator
catch ( NoDataSourceException noDataException ) {
    noDataException.printStackTrace();
}

// device connection error
catch ( IOException ioException ) {
    ioException.printStackTrace();
}

} // end method actionPerformed

} // end inner class CaptureHandler

// set output format of device-captured media
public void setDeviceFormat( Format currentFormat )
{
    // set desired format through all format controls
    for ( int i = 0; i < formatControls.length; i++ ) {
        // make sure format control is configurable
        if ( formatControls[ i ].isEnabled() ) {

            formatControls[ i ].setFormat( currentFormat );

            System.out.println (
                "Presentation output format currently set as " +
                formatControls[ i ].getFormat() );

        }

    } // end for loop

} // end method setDeviceFormat

// get selected device vector index
public int getSelectedDeviceIndex( String[] names )
{

Fig. 22.2 Formatting and saving media from capture devices (part 4 of 9).
// get device name from dialog box of device choices
String name = (String) JOptionPane.showInputDialog(this, "Select a device:", "Device Selection",
JOptionPane.QUESTION_MESSAGE,
null, names, names[0]);

// if format selected, get index of name in array names
if (name != null)
    return Arrays.binarySearch(names, name);

// else return bad selection value
else
    return -1;

// return user-selected format for device
public Format getSelectedFormat(Format[] showFormats)
{
    return (Format) JOptionPane.showInputDialog(this, "Select a format:", "Format Selection",
JOptionPane.QUESTION_MESSAGE,
null, showFormats, null);
}

// pop up error messages
public void showErrorMessage(String error)
{
    JOptionPane.showMessageDialog(this, error, "Error",
JOptionPane.ERROR_MESSAGE);
}

// get desired file for saved captured media
public File getSaveFile()
{
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    int result = fileChooser.showSaveDialog(this);

    if (result == JFileChooser.CANCEL_OPTION)
        return null;

    else
        return fileChooser.getSelectedFile();

    // show saving monitor of captured media
    public void showSaveMonitor()
    {

Fig. 22.2 Formatting and saving media from capture devices (part 5 of 9).
247       // show saving monitor dialog
248       int result = JOptionPane.showConfirmDialog( this,
249             saveProgress, "Save capture in progress...",
250             JOptionPane.DEFAULT_OPTION,
251             JOptionPane.INFORMATION_MESSAGE );
252
253       // terminate saving if user presses "OK" or closes dialog
254       if ( ( result == JOptionPane.OK_OPTION ) ||
255             ( result == JOptionPane.CLOSED_OPTION ) ) {
256           processor.stop();
257           processor.close();
258
259           System.out.println ( "Capture closed." );
260       }
261
262       // process captured media and save to file
263       public void captureSaveFile()
264       {
265           // array of desired saving formats supported by tracks
266           Format outFormats[] = new Format[ 1 ];
267
268           outFormats[ 0 ] = selectedFormat;
269
270           // file output format
271           FileTypeDescriptor outFileType =
272               new FileTypeDescriptor( FileTypeDescriptor.QUICKTIME );
273
274           // set up and start processor and monitor capture
275           try {
276               // create processor from processor model
277               // of specific data source, track output formats,
278               // and file output format
279               processor = Manager.createRealizedProcessor( 280                     new ProcessorModel( inSource, outFormats,
281                     outFileType ) );
282
283               // try to make a data writer for media output
284               if ( !makeDataWriter() )
285                   return;
286
287               // call start on processor to start captured feed
288               processor.start();
289
290               // get monitor control for capturing and encoding
291               MonitorControl monitorControl =
292                   ( MonitorControl ) processor.getControl( 293                       "javax.media.control.MonitorControl" );
294
295               // get GUI component of monitoring control
296               saveProgress = monitorControl.getControlComponent();
297
298           }
299
299         }

Fig. 22.2 Formatting and saving media from capture devices (part 6 of 9).
showSaveMonitor();

}  // end try

// no processor could be found for specific
// data source
catch ( NoProcessorException processorException ) {
    processorException.printStackTrace();
}

// unable to realize through
// createRealizedProcessor method
catch ( CannotRealizeException realizeException ) {
    realizeException.printStackTrace();
}

// device connection error
catch ( IOException ioException ) {
    ioException.printStackTrace();
}

}  // end method captureSaveFile

// method initializing media file writer
public boolean makeDataWriter()
{
    File saveFile = getSaveFile();

    if ( saveFile == null )
        return false;

    // get output data source from processor
    outSource = processor.getDataOutput();

    if ( outSource == null ) {
        showErrorMessage( "No output from processor!" );
        return false;
    }

    // start data writing process
    try {

        // create new MediaLocator from saveFile URL
        MediaLocator saveLocator =
            new MediaLocator ( saveFile.toURL() );

        // create DataSink from output data source
        // and user-specified save destination file
        dataSink = Manager.createDataSink(
            outSource, saveLocator );

Fig. 22.2  Formatting and saving media from capture devices (part 7 of 9).
// register a DataSinkListener for DataSinkEvents
dataSink.addDataSinkListener(

    // anonymous inner class to handle DataSinkEvents
    new DataSinkListener () {

        // if end of media, close data writer
        public void dataSinkUpdate(DataSinkEvent dataEvent) {
            // if capturing stopped, close DataSink
            if (dataEvent instanceof EndOfStreamEvent) {
                dataSink.close();
            }
        }
    } // end DataSinkListener
    ); // end call to method addDataSinkListener

    // start saving
    dataSink.open();
    dataSink.start();

} // end try

// DataSink could not be found for specific
// save file and data source
catch (NoDataSinkException noDataSinkException) {
    noDataSinkException.printStackTrace();
    return false;
}

// violation while accessing MediaLocator
// destination
catch (SecurityException securityException) {
    securityException.printStackTrace();
    return false;
}

// problem opening and starting DataSink
catch (IOException ioException) {
    ioException.printStackTrace();
    return false;
}

return true;

} // end method makeDataWriter

// main method
public static void main( String args[] ) {
    CapturePlayer testPlayer = new CapturePlayer();
```java
testPlayer.setSize( 200, 70 );
testPlayer.setLocation( 300, 300 );
testPlayer.setDefaultCloseOperation( EXIT_ON_CLOSE );
testPlayer.setVisible( true );
}
```

// end class CapturePlayer

Fig. 22.2  Formatting and saving media from capture devices (part 9 of 9).
Class **CapturePlayer** (Fig. 22.2) illustrates capturing, setting media formats and saving media from capture devices supported by JMF. The simplest test of the program uses a microphone for audio input. Initially, the GUI has only one button, which users click to begin the configuration process. Users then select a capture device from a pull-down menu dialog. The next dialog box has options for the format of the capture device and file output. The third dialog box asks users to save the media to a specific file. The final dialog box provides a volume control and the option of monitoring the data. Monitoring allows users to hear or see the media as it is captured and saved without modifying it in any way. Many media capture technologies offer monitoring capabilities. For example, many video recorders have a screen attachment to let users see what the camera is capturing without looking through the viewfinder. In a recording studio producers can listen to live music being recorded through headphones in another room. Monitoring data is different from playing data in that it does not make any changes to the format of the media or affect the data being sent to the **Processor**.

In the **CapturePlayer** program, lines 14–16 import the JMF Java extension packages `javax.media.protocol`, `javax.media.format` and `javax.media.control`, which contain classes and interfaces for media control and device formatting. Line 17 imports the JMF package `javax.media.datasink`, which contains classes for outputting formatted media. The program uses the classes and interfaces provided by these packages to obtain the desired capture device information, set the format of the capture device, create a **Processor** to handle the captured media data, create a **DataSink** to write the media data to a file and monitor the saving process.

**CapturePlayer** is capable of setting the media format. JMF provides class **Format** to describe the attributes of a media format, such as the **sampling rate** (which controls the quality of the sound) or whether the media should be in **stereo** or **mono** format. Each media format is encoded differently and can be played only with a media handler that supports its particular format. Line 28 declares an array of the **Formats** that the capture device supports and a **Format** reference for the user-selected format (**selectedFormat**).

After obtaining the **Format** objects, the program needs access to the formatting controls of the capture device. Line 31 declares an array to hold the **FormatControls** which will set the capture-device format. Class **CapturePlayer** sets the desired **Format** for the media source through the device’s **FormatControls** (line 31). The **CaptureDeviceInfo** reference **deviceInfo** (line 34) stores the capture device information, which will be placed in a **Vector** containing all of the device’s information.

Class **DataSource** connects programs to media data sources, such as capture devices. The **SimplePlayer** of Figure 22.1 accessed a **DataSource** object by invoking **Manager** method **createPlayer**, passing in a **MediaLocator**. However, class **CapturePlayer** accesses the **DataSource** directly. Line 40 declares two **DataSources**—**inSource** connects to the capture device’s media and **outSource** connects to the output data source to which the captured media will be saved.

An object that implements interface **Processor** provides the primary function that controls and processes the flow of media data in class **CapturePlayer** (line 46). The class also creates an object that implements interface **DataSink** to write the captured data to a file (line 43).

Clicking the **Capture and Save File** button configures the capture device by invoking method **actionPerformed** (lines 93–171) in **private** inner class **Cap-
tureHandler (lines 90–173). An instance of inner class CaptureHandler is registered to listen for ActionEvents from captureButton (line 62). The program provides users with a list of available capture devices when lines 96–97 invoke CaptureDeviceManager static method getDeviceList. Method getDeviceList obtains all of the computer’s available devices that support the specified Format. Specifying null as the Format parameter returns a complete list of available devices. Class CaptureDeviceManager enables a program to access this list of devices.

Lines 109–119 copy the names of all capture devices into a String array (deviceNames) for display purposes. Lines 122-123 invoke CapturePlayer method getSelectedDeviceIndex (lines 195–210) to show a selector dialog with a list of all the device names stored in array deviceNames. The method call to showInputDialog (lines 198–201) has a different parameter list than earlier examples. The first four parameters are the dialog’s parent component, message, title, and message type, as earlier chapters use. The final three, which are new, specify the icon (in this case, null), the list of values presented to the user (deviceNames), and the default selection (the first element of deviceNames). Once users select a device, the dialog returns the string, which is used to return the integer index of the selected name in deviceNames. This String helps determine the parallel element in the deviceList. This element, which is an instance of CaptureDeviceInfo, creates and configures a new device from which the desired media can be recorded.

A CaptureDeviceInfo object encapsulates the information that the program needs to access and configure a capture device, such as location and format preferences. Calling methods getLocator (line 143) and getFormats (line 132) access these pieces of information. Lines 129–130 access the new CaptureDeviceInfo that the user specified in the deviceList. Next, lines 135–136 call inSource’s disconnect method to disengage any previously opened capture devices before connecting the new device.

Lines 142–143 invoke Manager method createDataSource to obtain the DataSource object that connects to the capture device’s media source, passing in the capture device’s MediaLocator object as an argument. CaptureDeviceInfo method getLocator returns the capture device’s MediaLocator. Method createDataSource in turn invokes DataSource method connect which establishes a connection with the capture device. Method createDataSource throws a NoDataSourceException if it cannot locate a DataSource for the capture device. An IOException occurs if there is an error opening the device.

Before capturing the media, the program needs to format the DataSource as specified by the user in the Format Selection dialog. Lines 146–147 use CaptureDevice method getFormatControls to obtain an array of FormatControls for DataSource inSource. An object that implements interface FormatControl specifies the format of the DataSource. DataSource objects can represent media sources other than capture devices, so for this example the cast operator in line 146 manipulates object inSource as a CaptureDevice and accesses capture device methods such as getFormatControls. Line 150 invokes method getSelectedFormat (lines 213–219) to display an input dialog from which users can select one of the available formats. Lines 176–192 call the method setDeviceFormat to set the media output format for the device to the user-selected Format. Each capture device can have several FormatControls, so setDeviceFormat uses FormatControl method setFormat to specify the format for each FormatControl object.
Formatting the DataSource completes the configuration of the capture device. A Processor (object inSource) converts the data to the file format in which it will be saved. The Processor works as a connector between the outSource and method captureSaveFile since DataSource inSource does not play or save the media, it only serves to configure the capture device. Line 157 invokes method captureSaveFile (lines 265–322) to perform the steps needed to save the captured media in a recognizable file format.

To create a Processor, this program first creates a ProcessorModel, a template for the Processor. The ProcessorModel determines the attributes of a Processor through a collection of information which includes a DataSource or MediaLocator, the desired formats for the media tracks which the Processor will handle, and a ContentDescriptor indicating the output content type. Line 268 creates a new Format array (outFormats) that represents the possible formats for each track in the media. Line 270 sets the default format to the first element of the array. To save the captured output to a file, the Processor must first convert the data it receives to a file-enabled format. A new QuickTime FileTypeDescriptor (package javax.media.format) is created to store a description of the content type of the Processor’s output and store it in outFileType (lines 273–274). Lines 282–284 use the DataSource inSource, the array outFormats, and the file type outFileType to instantiate a new ProcessorModel (lines 283–284).

Generally, Processors need to be configured before they can process media, but the one in this application does not since lines 282–284 invoke Manager method createRealizedProcessor. This method creates a configured, realized Processor based on the ProcessorModel object passed in as an argument. Method createRealizedProcessor throws a NoProcessorException if the program cannot locate a Processor for the media or if JMF does not support the media type. The method throws a CannotRealizeException if the Processor cannot be realized. This may occur if another program is already using the media, thus blocking communication with the media source.

**Common Programming Error 22.1**

Be careful when specifying track formats. Incompatible formats for specific output file types prevent the program from realizing the Processor.

**Software Engineering Observation 22.2**

Recall that the Processor transitions through several states before being realized. Manager method createProcessor allows a program to provide more customized configuration before a Processor is realized.

**Performance Tip 22.2**

When method createRealizedProcessor configures the Processor, the method blocks until the Processor is realized. This may prevent other parts of the program from executing. In some cases, using a ControllerListener to respond to Controller-Events may enable a program to operate more efficiently. When the Processor is realized, the listener is notified so the program can begin processing the media.

Having obtained media data in a file format from the Processor, the program can make a “data writer” to write the media output to a file. An object that implements interface DataSink enables media data to be output to a specific location—most commonly a file.
Line 287 invokes method `makeDataWriter` (lines 325–399) to create a `DataSink` object that can save the file. `Manager` method `createDataSink` requires the `DataSource` of the `Processor` and the `MediaLocator` for the new file as arguments. Within `makeDataWriter`, lines 229–242 invokes method `getSaveFile` to prompt users to specify the name and location of the file to which the media should be saved. The `File` object `saveFile` stores the information. `Processor` method `getDataOutput` (line 333) returns the `DataSource` from which it received the media. Lines 344–345 create a new `MediaLocator` for `saveFile`. Using this `MediaLocator` and the `DataSource`, lines 349–350 create a `DataSink` object which writes the output media from the `DataSource` to the file in which the data will be saved, as specified by the `MediaLocator`. Method `createDataSink` throws a `NoDataSinkException` if it cannot create a `DataSink` that can read data from the `DataSource` and output the data to the location specified by the `MediaLocator`. This failure may occur as a result of invalid media or an invalid `MediaLocator`.

The program needs to know when to stop outputting data, so lines 353–369 register a `DataSinkListener` to listen for `DataSinkEvent`s. `DataSinkListener` method `dataSinkUpdate` (lines 359–365) is called when each `DataSinkEvent` occurs. If the `DataSinkEvent` is an `EndOfStreamEvent`, indicating that the `Processor` has been closed because the capture stream connection has closed, line 364 closes the `DataSink`. Invoking `DataSink` method `close` stops the data transfer. A closed `DataSink` cannot be used again.

**Common Programming Error 22.2**

The media file output with a DataSink will be corrupted if the DataSink is not closed properly.

**Software Engineering Observation 22.3**

Captured media may not yield an EndOfMediaEvent if the media’s endpoint cannot be determined.

After setting up the `DataSink` and registering its listener, line 372 calls `DataSink` method `open` to connect the `DataSink` to the destination that the `MediaLocator` specifies. Method `open` throws a `SecurityException` if the `DataSink` attempts to write to a destination for which the program does not have write permission, such as a read-only file.

Line 373 calls `DataSink` method `start` to initiate data transfer. At this point, the program returns from method `makeDataWriter` back to method `captureSaveFile` (lines 265–322). Although the `DataSink` prepares itself to receive the transfer and indicates that it is ready by calling `start`, the transfer does not actually take place until the `Processor`’s `start` method is called. The invocation of `Processor` method `start` begins the flow of data from the capture device, formats the data and transfers that data to the `DataSink`. The `DataSink` writes the media to a file, which completes the process performed by class `CapturePlayer`.

While `Processor` encodes the data and the `DataSink` saves it to a file, `CapturePlayer` monitors the process. Monitoring provides a method of overseeing data as the capture device collects it. Lines 294–296 obtain an object that implements interface `MonitorControl` (package `javax.media.control`) from the `Processor` by calling method `getControl`. Line 299 calls `MonitorControl` method `getControlCom-
ponent to obtain the GUI component that displays the monitored media. MonitorControls typically have a checkbox to enable or disable displaying media. Also, audio devices have a volume control and video devices have a control for setting the preview frame rate. Line 301 invokes method showSaveMonitor (lines 245–261) to display the monitoring GUI in a dialog box. To terminate capturing, users can press the OK button or close the dialog box (lines 254–261). For some video capture devices, the Processor needs to be in a Stopped state to enable monitoring of the saving and capturing process.

Thus far, we have discussed JMF’s capabilities to access, present and save media content. Our final JMF example demonstrates how to send media between computers using JMF’s streaming capabilities.

### 22.4 RTP Streaming

Streaming media refers to media that is transferred from a server to a client in a continuous stream of bytes. The client can begin playing the media while still downloading the media from the server. Audio and video media files are often many megabytes in size. Live events, such as concerts or football game broadcasts, may have indeterminable sizes. Users could wait until a recording of a concert or game is posted, then download the entire recording. However, at today’s Internet connection speeds, downloading such a broadcast could take days and typically users prefer to listen to live broadcasts as they occur. Streaming media enables client applications to play media over the Internet or over a network without downloading the entire media file at once.

In a streaming media application, the client typically connects to a server that sends a stream of bytes containing the media back to the client. The client application buffers (i.e. stores locally) a portion of the media, which the client begins playing after a certain portion has been received. The client continually buffers additional media, providing users with an uninterrupted clip, as long as network traffic does not prevent the client application from buffering additional bytes. With buffering, users experience the media within seconds of when the streaming begins, even though all of the material has not been received.

**Performance Tip 22.3**

Streaming media to a client enables the client to experience the media faster than if the client must wait for an entire media file to download.

Demand for real-time, robust multimedia is rising dramatically as the speed of Internet connections increase. Broadband Internet access, which provides high-speed network connections to the Internet for so many home users, is becoming more popular, though the number of users remains relatively small compared to the total number of Internet users. With faster connections, streaming media can provide a better multimedia experience. Users with slower connections can still experience the multimedia, though with lesser quality. The wide range of applications that use streaming media is growing. Applications that stream video clips to clients have expanded to provide real-time broadcast feeds. Thousands of radio stations stream music continuously over the Internet. Client applications such as RealPlayer have focused on streaming media content with live radio broadcasts. Applications are not limited to audio and video server-to-client streaming. For example, teleconferencing and video conferencing applications increase efficiency in everyday business by reducing the need for business people to travel great distances to attend meetings.
JMF provides a streaming media package that enables Java applications to send and receive streams of media in some of the formats discussed earlier in this chapter. For a complete list of formats, see the official JMF Web site:

java.sun.com/products/java-media/jmf/2.1.1/formats.html

JMF uses the industry-standard *Real-Time Transport Protocol (RTP)* to control media transmission. RTP is designed specifically to transmit real-time media data.

The two mechanisms for streaming RTP-supported media are passing it through a `DataSink` and buffering it. The easier mechanism to use is a `DataSink`, which writes the contents of a stream to a host destination (i.e., a client computer), via the same techniques shown in Fig. 22.2 to save captured media to a file. In this case, however, the destination `MediaLocator`'s URL would be specified in the following format:

`rtp://host:port/contentType`

where `host` is the IP address or host name of the server, `port` is the port number on which the server is streaming the media and `contentType` is either `audio` or `video`.

Using a `DataSink` as specified allows only one stream to be sent at a time. To send multiple streams (e.g., as a karaoke video with separate tracks for video and audio would) to multiple hosts, a server application must use *RTP session managers*. An `RTPManager` (package `javax.media.rtp`) provides more control over the streaming process, permitting specification of buffer sizes, error checking and streaming reports on the *propagation delay* (the time it takes for the data to reach its destination).

The program in Fig. 22.3 and Fig. 22.4 demonstrates streaming using the RTP session manager. This example supports sending multiple streams in parallel, so separate clients must be opened for each stream. This example does not show a client that can receive the RTP stream. The program in Fig. 22.1 (`SimplePlayer`) can test the RTP server by specifying an RTP session address

`rtp://127.0.0.1:4000/audio`

as the location `SimplePlayer` should open to begin playing audio. To execute the streaming media server on a different computer, replace `127.0.0.1` with either the IP address or host name of the server computer.

```java
// Fig. 22.3: RTPServer.java
// Provides configuration and sending capabilities
// for RTP-supported media files

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

// Java extension packages
import javax.media.*;
import javax.media.protocol.*;
import javax.media.control.***;
import javax.media.rtp.*;
import javax.media.format.*;
```

**Fig. 22.3**  Serving streaming media with RTP session managers (part 1 of 7).
public class RTPServer {
    // IP address, file or medialocator name, port number
    private String ipAddress, fileName;
    private int port;

    // processor controlling data flow
    private Processor processor;

    // data output from processor to be sent
    private DataSource outSource;

    // media tracks' configurable controls
    private TrackControl tracks[];

    // RTP session manager
    private RTPManager rtpManager[];

    // constructor for RTPServer
    public RTPServer( String locator, String ip, int portNumber )
    {
        fileName = locator;
        port = portNumber;
        ipAddress = ip;
    }

    // initialize and set up processor
    // return true if successful, false if not
    public boolean beginSession()
    {
        // get MediaLocator from specific location
        MediaLocator mediaLocator = new MediaLocator( fileName );

        if ( mediaLocator == null ) {
            System.err.println(  
                "No MediaLocator found for " + fileName );
            return false;
        }

        try {
            processor = Manager.createProcessor( mediaLocator );
            // register a ControllerListener for processor
            // to listen for state and transition events
            processor.addControllerListener(  
                new ProcessorEventHandler() );

            System.out.println( "Processor configuring..." );
        }
    }
}

Fig. 22.3 Serving streaming media with RTP session managers (part 2 of 7).
// configure processor before setting it up
processor.configure();

// source connection error
catch ( IOException ioException ) {
    ioException.printStackTrace();
    return false;
}

// exception thrown when no processor could
// be found for specific data source
catch ( NoProcessorException noProcessorException ) {
    noProcessorException.printStackTrace();
    return false;
}

return true;

}  // end method beginSession

// ControllerListener handler for processor
private class ProcessorEventHandler
    extends ControllerAdapter {

    // set output format and realize
    // configured processor
    public void configureComplete(
        ConfigureCompleteEvent configureCompleteEvent )
    {
        System.out.println( "Processor configured." );
        setOutputFormat();
        System.out.println( "Realizing Processor...\n" );
        processor.realize();
    }

    // start sending when processor is realized
    public void realizeComplete(
        RealizeCompleteEvent realizeCompleteEvent )
    {
        System.out.println( "\nInitialization successful for " + fileName );
        if ( transmitMedia() == true )
            System.out.println( "\nTransmission setup OK" );
        else
            System.out.println( "\nTransmission failed." );
    }

Fig. 22.3 Serving streaming media with RTP session managers (part 3 of 7).
```java
120    // stop RTP session when there is no media to send
121    public void endOfMedia( EndOfMediaEvent mediaEndEvent )
122    {
123        stopTransmission();
124        System.out.println ( "Transmission completed." );
125    }
126
127    }  // end inner class ProcessorEventHandler
128
129    // set output format of all tracks in media
130    public void setOutputFormat()
131    {
132        // set output content type to RTP capable format
133        processor.setContentDescriptor(
134            new ContentDescriptor( ContentDescriptor.RAW_RTP ) );
135
136        // get all track controls of processor
137        tracks = processor.getTrackControls();
138
139        // supported RTP formats of a track
140        Format rtpFormats[];
141
142        // set each track to first supported RTP format
143        // found in that track
144        for ( int i = 0; i < tracks.length; i++ ) {
145
146            System.out.println( "\nTrack #" +
147                ( i + 1 ) + " supports ");
148
149            if ( tracks[ i ].isEnabled() ) {
150
151                rtpFormats = tracks[ i ].getSupportedFormats();
152
153                // if supported formats of track exist,
154                // display all supported RTP formats and set
155                // track format to be first supported format
156                if ( rtpFormats.length > 0 ) {
157
158                    for ( int j = 0; j < rtpFormats.length; j++ )
159                        System.out.println( rtpFormats[ j ] );
160
161                    tracks[ i ].setFormat( rtpFormats[ 0 ] );
162
163                    System.out.println ( "Track format set to " +
164                        tracks[ i ].getFormat() );
165                }
166
167                else
168                    System.err.println ( "No supported RTP formats for track!" );
169            }
170
171        }  // end if
172```

Fig. 22.3 Serving streaming media with RTP session managers (part 4 of 7).
send media with boolean success value

```java
public boolean transmitMedia()
{
    outSource = processor.getDataOutput();
    if ( outSource == null ) {
        System.out.println ( "No data source from media!" );
        return false;
    }
    rtpManager = new RTPManager[ tracks.length ];
    SessionAddress localAddress, remoteAddress;

    SendStream sendStream;
    InetAddress ip;

    try {
        // transmit every track in media
        for ( int i = 0; i < tracks.length; i++ ) {
            // instantiate a RTPManager
            rtpManager[ i ] = RTPManager.newInstance();
            // add 2 to specify next control port number;
            // (RTP Session Manager uses 2 ports)
            port += ( 2 * i );
            // get IP address of host from ipAddress string
            ip = InetAddress.getByName( ipAddress );
            // encapsulate pair of IP addresses for control and
            // data with 2 ports into local session address
            localAddress = new SessionAddress( ip.getLocalHost(), port );
            // get remoteAddress session address
            remoteAddress = new SessionAddress( ip, port );
            // initialize the session
            rtpManager[ i ].initialize( localAddress );
```

Fig. 22.3 Serving streaming media with RTP session managers (part 5 of 7).
// open RTP session for destination
rtpManager[ i ].addTarget( remoteAddress );

System.out.println( "\nStarted RTP session: " + ipAddress + " " + port); // create send stream in RTP session
sendStream = rtpManager[ i ].createSendStream( outSource, i );

// start sending the stream
sendStream.start();

System.out.println( "Transmitting Track #" + ( i + 1 ) + " ...");

// start media feed
processor.start();

} // end try

// unknown local or unresolvable remote address
catch ( InvalidSessionAddressException addressError ) {
    addressError.printStackTrace();
    return false;
}

// DataSource connection error
catch ( IOException ioException ) {
    ioException.printStackTrace();
    return false;
}

// format not set or invalid format set on stream source
catch ( UnsupportedFormatException formatException ) {
    formatException.printStackTrace();
    return false;
}

// transmission initialized successfully
return true;

} // end method transmitMedia

// stop transmission and close resources
public void stopTransmission() {
    if ( processor != null ) {

Fig. 22.3  Serving streaming media with RTP session managers (part 6 of 7).
Class **RTPServer**’s purpose is to stream media content. As in previous examples, the **RTPServer** (Fig. 22.3) sets up the media, processes and formats it, then outputs it. **ControllerEvent**s and the various states of the streaming process drive this process. Processing of the media has three distinct parts—**Processor** initialization, **Format** configuration and data transmission. The code in this example contains numerous confirmation messages displayed to the command prompt and an emphasis on error checking. A problem during streaming will most likely end the entire process and it will need to be restarted.

To test **RTPServer**, class **RTPServerTest** (Fig. 22.4) creates a new **RTPServer** object and passes its constructor (lines 35–40) three arguments—a **String** representing the media’s location, a **String** representing the IP address of the client and a port number for streaming content. These arguments contain the information class **RTPServer** needs to obtain the media and set up the streaming process. Following the general approach outlined in **SimplePlayer** (Fig. 22.1) and **CapturePlayer** (Fig. 22.2), class **RTPServer** obtains a media source, configures the source through a type of **Controller** and outputs the data to a specified destination.

**RTPServerTest** calls **RTPServer** method **beginSession** (lines 44–86) to set up the **Processor** that controls the data flow. Line 47 creates a **MediaLocator** and initializes it with the media location stored in **fileName**. Line 58 creates a **Processor** for the data specified by that **MediaLocator**.
Unlike the program in Fig. 22.2, this Processor is not preconfigured by a Manager. Until class RTPServer configures and realizes the Processor, the media cannot be formatted. Lines 62–63 register a ProcessorEventHandler to react to the processor’s ControllerEvents. The methods of class ProcessorEventHandler (lines 89–127) control the media setup as the Processor changes states. Line 68 invokes Processor method configure to place the Processor in the Configuring state. Configuration is when the Processor inquires of the system and the media the format information needed to program the Processor to perform the correct task. A ConfigureCompleteEvent occurs when the Processor completes configuration. The ProcessorEventHandler method configureComplete (lines 94–104) responds to this transition. Method configureComplete calls method setOutputFormat (lines 130–175) then realizes the Processor (line 103). When line 99 invokes method setOutputFormat, it sets each media track to an RTP streaming media format. Lines 133–134 in method setOutputFormat specify the output content-type by calling Processor method setContentDescriptor. The method takes as an argument a ContentDescriptor initialized with constant ContentDescriptor.RAW_RTP. The RTP output content-type restricts the Processor to only support RTP-enabled media track formats. The Processor’s output content-type should be set before the media tracks are configured.

Once the Processor is configured, the media track formats need to be set. Line 137 invokes Processor method getTrackControls to obtain an array that contains the corresponding TrackControl object (package javax.media.control) for each media track. For each enabled TrackControl, lines 144–173 obtain an array of all supported RTP media Formats (line 151), then set the first supported RTP format as the preferred RTP-streaming media format for that track (line 161). When method setOutputFormat returns, line 103 in method configureComplete realizes the Processor.

As with any controller realization, the Processor can output media as soon as it has finished realizing itself. When the Processor enters the Realized state, the ProcessorEventHandler invokes method realizeComplete (lines 107–118). Line 113 invokes method transmitMedia (lines 178–271) which creates the structures needed to transmit the media to the Processor. This method obtains the DataSource from the Processor (line 180), then declares an array of RTPManagers that are able to start and control an RTP session (line 189). RTPManagers use a pair of SessionAddress objects with identical IP addresses, but different port numbers—one for stream control and one for streaming media data. An RTPManager receives each IP address and port number as a SessionAddress object. Line 192 declares the SessionAddress objects used in the streaming process. An object that implements interface SendStream (line 195) performs the RTP streaming.

Software Engineering Observation 22.4

For videos that have multiple tracks, each SendStream must have its own RTPManager managing its session. Each Track has its own SendStream.

The try block (lines 201–248) of method transmitMedia sends out each track of the media as an RTP stream. First, managers must be created for the sessions. Line 207 invokes RTPManager method newInstance to instantiate an RTPManager for each
track stream. Line 211 assigns the port number to be two more than the previous port number, because the each track uses one port number for the stream control and one to actually stream the data. Lines 218–219 instantiate a new local session address where the stream is located (i.e., the RTP address that clients use to obtain the media stream) with the local IP address and a port number as parameters. Line 219 invokes InetAddress method getLocalHost to get the local IP address. Line 222 instantiates the client’s session address, which the RTPManager uses as the stream’s target destination. When line 225 calls RTPManager method initialize, the method directs the local streaming session to use the local session address. Using object remoteAddress as a parameter, line 228 calls RTPManager method addTarget to open the destination session at the specified address. To stream media to multiple clients, call RTPManager method addTarget for each destination address. Method addTarget must be called after the session is initialized and before any of the streams are created on the session.

The program can now create the streams on the session and start sending data. The streams are created in the current RTP session with the DataSource outSource (obtained at line 180) and the source stream index (i.e. media track index) in lines 234–235. Invoking method start on the SendStream (line 238) and on the Processor (line 246) starts transmission of the media streams, which may cause exceptions. An InvalidSessionAddressException occurs when the specified session address is invalid. An UnsupportedFormatException occurs when an unsupported media format is specified or if the DataSource’s Format has not been set. An IOException occurs if the application encounters networking problems. During the streaming process, RTPManagers can be used with related classes in package javax.media.rtp and package javax.media.rtp.event controls the streaming process and send reports to the application.

The program should close connections and stop streaming transmission when it reaches the end of the streaming media or when the program terminates. When the Processor encounters the end of the media, it generates an EndOfMediaEvent. In response, the program calls method endOfMedia (lines 121–125). Line 123 invokes method stopTransmission (lines 274–303) to stop and close the Processor (lines 279–282). After calling stopTransmission, streaming cannot resume because it disposes of the Processor and the RTP session resources. Lines 288–297 invoke RTPManager method removeTargets (lines 292–293) to close streaming to all destinations. RTPManager method dispose (line 296) is also invoked, releasing the resources held by the RTP sessions. The class RTPServerTest (Fig. 22.4), explicitly invokes method stopTransmission when users terminates the server application (line 40).

```
1 // Fig. 22.4: RTPServerTest.java
2 // Test class for RTPServer
3
4 // Java core packages
5 import java.awt.event.*;
6 import java.io.*;
7 import java.net.*;
8```

Fig. 22.4 Application to test class RTPServer from Fig. 22.3 (part 1 of 6).
// Java extension packages
import javax.swing.*;

public class RTPServerTest extends JFrame {

    // object handling RTP streaming
    private RTPServer rtpServer;

    // media sources and destination locations
    private int port;
    private String ip, mediaLocation;
    private File mediaFile;

    // GUI buttons
    private JButton transmitFileButton, transmitUrlButton;

    // constructor for RTPServerTest
    public RTPServerTest() {
        super( "RTP Server Test" );

        // register a WindowListener for frame events
        addWindowListener(
            // anonymous inner class to handle WindowEvents
            new WindowAdapter() {
                
                public void windowClosing(
                    WindowEvent windowEvent )
                {
                    if ( rtpServer != null )
                        rtpServer.stopTransmission();
                }
            } // end WindowAdapter
        ); // end call to method addWindowListener

        // panel containing button GUI
        JPanel buttonPanel = new JPanel();
        getContentPane().add( buttonPanel );

        // transmit file button GUI
        transmitFileButton = new JButton( "Transmit File" );
        buttonPanel.add( transmitFileButton );

        // register ActionListener for transmitFileButton events
        transmitFileButton.addActionListener(
            new ButtonHandler() );

        // transmit URL button GUI
        transmitUrlButton = new JButton( "Transmit Media" );
        buttonPanel.add( transmitUrlButton );

    }

    private RTPServer rtpServer;

    // media sources and destination locations
    private int port;
    private String ip, mediaLocation;
    private File mediaFile;

    // GUI buttons
    private JButton transmitFileButton, transmitUrlButton;

    // constructor for RTPServerTest
    public RTPServerTest() {
        super( "RTP Server Test" );

        // register a WindowListener for frame events
        addWindowListener(
            // anonymous inner class to handle WindowEvents
            new WindowAdapter() {
                
                public void windowClosing(
                    WindowEvent windowEvent )
                {
                    if ( rtpServer != null )
                        rtpServer.stopTransmission();
                }
            } // end WindowAdapter
        ); // end call to method addWindowListener

        // panel containing button GUI
        JPanel buttonPanel = new JPanel();
        getContentPane().add( buttonPanel );

        // transmit file button GUI
        transmitFileButton = new JButton( "Transmit File" );
        buttonPanel.add( transmitFileButton );

        // register ActionListener for transmitFileButton events
        transmitFileButton.addActionListener(
            new ButtonHandler() );

        // transmit URL button GUI
        transmitUrlButton = new JButton( "Transmit Media" );
        buttonPanel.add( transmitUrlButton );

    } // end constructor

    private RTPServer rtpServer;

    // media sources and destination locations
    private int port;
    private String ip, mediaLocation;
    private File mediaFile;

    // GUI buttons
    private JButton transmitFileButton, transmitUrlButton;

    // constructor for RTPServerTest
    public RTPServerTest() {
        super( "RTP Server Test" );

        // register a WindowListener for frame events
        addWindowListener(
            // anonymous inner class to handle WindowEvents
            new WindowAdapter() {
                
                public void windowClosing(
                    WindowEvent windowEvent )
                {
                    if ( rtpServer != null )
                        rtpServer.stopTransmission();
                }
            } // end WindowAdapter
        ); // end call to method addWindowListener

        // panel containing button GUI
        JPanel buttonPanel = new JPanel();
        getContentPane().add( buttonPanel );

        // transmit file button GUI
        transmitFileButton = new JButton( "Transmit File" );
        buttonPanel.add( transmitFileButton );

        // register ActionListener for transmitFileButton events
        transmitFileButton.addActionListener(
            new ButtonHandler() );

        // transmit URL button GUI
        transmitUrlButton = new JButton( "Transmit Media" );
        buttonPanel.add( transmitUrlButton );

    } // end constructor
// register ActionListener for transmitURLButton events
transmitUrlButton.addActionListener(
    new ButtonHandler());

// end constructor

// inner class ButtonHandler implements ActionListener {
private class ButtonHandler implements ActionListener {

    // open and try to send file to user-input destination
    public void actionPerformed( ActionEvent actionEvent ) {
        // if transmitFileButton invoked, get file URL string
        if ( actionEvent.getSource() == transmitFileButton ) {
            mediaFile = getFile();
            if ( mediaFile != null )
                try {
                    mediaLocation = mediaFile.toURL().toString();
                }
                catch ( MalformedURLException badURL ) {
                    badURL.printStackTrace();
                }
            else
                return;
        }
    // end if

        // else transmitMediaButton invoked, get location
        else
            mediaLocation = getMediaLocation();

        if ( mediaLocation == null )
            return;

    // get IP address
    ip = getIP();
    if ( ip == null )
        return;

    // get port number
    port = getPort();

Fig. 22.4 Application to test class RTPServer from Fig. 22.3 (part 3 of 6).
// check for valid positive port number and input
if ( port <= 0 ) {
    if ( port != -999 )
        System.err.println( "Invalid port number!" );
    return;
}

// instantiate new RTP streaming server
rtpServer = new RTPServer( mediaLocation, ip, port );
rtpServer.beginSession();

// get file from computer
public File getFile()
{
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
    int result = fileChooser.showOpenDialog( this );
    if ( result == JFileChooser.CANCEL_OPTION )
        return null;
    else
        return fileChooser.getSelectedFile();
}

// get media location from user
public String getMediaLocation()
{
    String input = JOptionPane.showInputDialog( this, "Enter MediaLocator" );
    if ( input != null && input.length() == 0 ) {
        System.err.println( "No input!" );
        return null;
    }
    return input;
}

// method getting IP string from user
public String getIP()
{
String input = JOptionPane.showInputDialog(
    this, "Enter IP Address: ");

    // if user presses OK with no input
if ( input != null && input.length() == 0 ) {
    System.err.println( "No input!" );
    return null;
}

    return input;

    // get port number
public int getPort() {
    String input = JOptionPane.showInputDialog(
        this, "Enter Port Number: ");

    // return flag value if user clicks OK with no input
if ( input != null && input.length() == 0 ) {
    System.err.println( "No input!" );
    return -999;
}

    // return flag value if user clicked CANCEL
if ( input == null )
    return -999;

    // else return input
    return Integer.parseInt( input );

} // end method getPort

    // execute application
public static void main( String args[] ) {
    RTPServerTest serverTest = new RTPServerTest();

    serverTest.setSize( 250, 70 );
    serverTest.setLocation( 300, 300 );
    serverTest.setDefaultCloseOperation( EXIT_ON_CLOSE );
    serverTest.setVisible( true );
}

} // end class RTPServerTest

Fig. 22.4 Application to test class RTPServer from Fig. 22.3 (part 5 of 6).
Many of today’s computer programs capture users’ attention with audio features. Even basic applets and applications can enhance users’ experiences with simple sounds or music clips. With sound programming interfaces, developers can create applications that play sounds in response to user interactions. For example, in many applications, when an error occurs and a dialog box appears on the screen, the dialog is often accompanied by a sound. Users thus receive both audio and visual indications that a problem has occurred. As another example, game programmers use extensive audio capabilities to enhance players’ experiences.
The Java Sound API is a simpler way to incorporate audio media into applications than the Java Media Framework. The Java Sound API is bundled with Java 2 Software Development Kit version 1.3. The API consists of four packages—javax.sound.midi, javax.sound.midi.spi, javax.sound.sampled and javax.sound.sampled.spi. The next two sections focus on packages javax.sound.midi and javax.sound.sampled, which provide classes and interfaces for accessing, manipulating and playing Musical Instrumental Data Interface (MIDI) and sampled audio. The packages ending in .spi provide developers with the tools to add Java Sound support for additional audio formats that are beyond the scope of this book.

The Java Sound API provides access to the Java Sound Engine which creates digitized audio and captures media from the supported sound devices discussed in Section 22.3. Java Sound requires a sound card to play audio. A program using Java Sound will throw an exception if it accesses audio system resources on a computer that does not have a sound card.

22.6 Playing Sampled Audio

This section introduces the features of package javax.sound.sampled for playing sampled audio file formats, which include Sun Audio (.au), Windows Waveform (.wav) and Macintosh Audio Interchange File Format (.aiff). The program in Fig. 22.5 and Fig. 22.6 shows how audio is played using these file formats.

When processing audio data, a line provides the path through which audio flows in a system. One example of a line is a pair of headphones connected to a CD player.

Class ClipPlayer (Fig. 22.5) is an example of how lines can be used. It contains an object that implements interface Clip, which in turn extends interface DataLine. A Clip is a line that processes an entire audio file rather than reading continuously from an audio stream. DataLines enhance Lines by providing additional methods (such as start and stop) for controlling the flow of data, and Clips enhance DataLines by providing methods for opening Clips and methods for precise control over playing and looping the audio.

```java
1 // Fig. 22.5: ClipPlayer.java
2 // Plays sound clip files of type WAV, AU, AIFF
3 // Java core packages
4 import java.io.*;
5
6 // Java extension packages
7 import javax.sound.sampled.*/;
8
9 public class ClipPlayer implements LineListener {
10   // audio input stream
11   private AudioInputStream soundStream;
12
13   // audio sample clip line
14   private Clip clip;
```

Fig. 22.5 ClipPlayer plays an audio file (part 1 of 4).
// Audio clip file
private File soundFile;

// boolean indicating replay of audio
private boolean replay = false;

// constructor for ClipPlayer
public ClipPlayer( File audioFile )
{
    soundFile = audioFile;
}

// open music file, returning true if successful
public boolean openFile()
{
    // get audio stream from file
    try {
        soundStream =
            AudioSystem.getAudioInputStream( soundFile );
    }

    // audio file not supported by JavaSound
    catch ( UnsupportedAudioFileException audioException ) {
        audioException.printStackTrace();
        return false;
    }

    // I/O error attempting to get stream
    catch ( IOException ioException ) {
        ioException.printStackTrace();
        return false;
    }

    // invoke loadClip, returning true if load successful
    return loadClip();
}

// load sound clip
public boolean loadClip ()
{
    // get clip line for file
    try {

        // get audio format of sound file
        AudioFormat audioFormat = soundStream.getFormat();

        // define line information based on line type,
        // encoding and frame sizes of audio file
        DataLine.Info dataLineInfo = new DataLine.Info( 
            Clip.class, AudioSystem.getTargetFormats( 
                AudioFormat.Encoding.PCM_SIGNED, audioFormat ),
        )
    }

Fig. 22.5  **ClipPlayer** plays an audio file (part 2 of 4).
audioFormat.getFrameSize(),
  audioFormat.getFrameSize() * 2 );

// make sure sound system supports data line
if ( !AudioSystem.isLineSupported(dataLineInfo) ) {
  System.err.println( "Unsupported Clip File!" );
  return false;
}

// get clip line resource
clip = ( Clip ) AudioSystem.getLine(dataLineInfo);

// listen to clip line for events
clip.addLineListener( this );

// open audio clip and get required system resources
clip.open(soundStream);

} // end try

// line resource unavailable
catch ( LineUnavailableException noLineException ) {
  noLineException.printStackTrace();
  return false;
}

// I/O error during interpretation of audio data
catch ( IOException ioException ) {
  ioException.printStackTrace();
  return false;
}

// clip file loaded successfully
return true;

} // end method loadClip

// start playback of audio clip
public void play() {
  clip.start();
}

// line event listener method to stop or replay at clip end
public void update( LineEvent lineEvent ) {
  // if clip reaches end, close clip
  if ( lineEvent.getType() == LineEvent.Type.STOP && !replay )
    close();

Fig. 22.5 ClipPlayer plays an audio file (part 3 of 4).
All Lines generate LineEvents, which can be handled by LineListeners. LineEvents occur when starting, stopping, playing and closing a Line object. Although a Line stops playback automatically when it reaches the end of an audio file, class ClipPlayer implements interface LineListener (line 10) and can close the Clip permanently or replay the Clip (discussed shortly). LineListeners are useful for tasks that must be synchronized with the LineEvent states of a line.

The Clip reads audio data from an AudioInputStream (a subclass of InputStream), which provides access to the stream’s data content. This example loads clips of the audio data before attempting to play it, and therefore is able to determine the length of the clip in frames. Each frame represents data at a specific time interval in the audio file. To play sampled audio files using Java Sound, a program must obtain an AudioInputStream from an audio file, obtain a formatted Clip line, load the AudioInputStream into the Clip line and start the data flow in the Clip line.

To play back sampled audio, the audio stream must be obtained from an audio file. ClipPlayer method openFile (lines 31–54) obtains audio from soundFile (initialized in the ClipPlayer constructor at lines 25–28). Lines 35–36 call AudioSystem static method getAudioInputStream to obtain an AudioInputStream for soundFile. Class AudioSystem facilitates access to many of the resources needed to play and manipulate sound files. Method getAudioInputStream throws an Unsup-
If the specified sound file is a non-audio file or if it contains a format that is not supported by Java Sound.

Next the program must provide a line through which audio data can be processed. Line 52 invokes method `loadClip` (lines 57–106) to open a `Clip` line and load the audio stream for playback. Line 81 invokes `AudioSystem static method getLine` to obtain a `Clip` line for audio playback. Method `getLine` requires a `Line.Info` object as an argument, to specify the attributes of the line that the `AudioSystem` should return. The line must be able to process audio clips of all supported sampled audio formats, so the `DataLine.Info` object must specify a `Clip` data line and a general encoding format. A buffer range should also be specified so the program can determine the best buffer size. The `DataLine.Info` constructor receives four arguments. The first two are the format (of type `AudioFormat.Encoding`) into which the program should convert the audio data and the `AudioFormat` of the audio source. The `AudioFormat` sets the format supported by the line, according to the audio format of the stream. Line 63 obtains the `AudioFormat` of the `AudioInputStream`, which contains format specifications that the underlying system uses to translate the data into sounds. Lines 68–69 call `AudioSystem method getTargetFormats` to obtain an array of the supported `AudioFormat`s. The third argument of the `DataLine.Info` constructor, which specifies the minimum buffer size, is set to the number of bytes in each frame of the audio stream. Line 70 invokes `AudioFormat method getFrameSize` to obtain the size of each frame in the audio stream. The maximum buffer size should be equivalent to two frames of the audio stream (line 71). Using the `DataLine.Info` object, line 74 checks if the underlying audio system supports the specified line. If it does, line 81 obtains the line from the audio system.

When an audio clip starts playing and when it finishes, the program needs to be alerted. Line 84 registers a `LineListener` for the `Clip`'s `LineEvents`. If a `LineEvent` occurs, the program calls `LineListener method update` (lines 115–133) to process it. The four `LineEvent` types, as defined in class `LineEvent.Type`, are OPEN, CLOSE, START and STOP. When the event type is `LineEvent.Type.STOP` and variable `replay` is `false`, line 120 calls `ClipPlayer`'s `close` method (lines 142–148) to stop audio playback and close the `Clip`. All audio resources obtained previously by the `Clip` are released when audio playback stops. When the event type is `LineEvent.Type.STOP` and variable `replay` is `true`, line 131 calls `Clip` method `loop` with parameter `Clip.LOOP_CONTINUOUSLY`, causing the `Clip` to loop until the user terminates the application. Invoking method `stop` of interface `Clip` stops data activity in the `Line`. Invoking method `start` resumes data activity.

Once the program finishes validating the `Clip`, line 87 calls `Clip method open` with the `AudioInputStream soundStream` as an argument. The `Clip` obtains the system resources required for audio playback. `AudioSystem method getLine` and `Clip method open` throw `LineUnavailableException` if another application is using the requested audio resource. `Clip method open` also throws an `IOException` if the `Clip` cannot read the specified `AudioInputStream`. When the test program (Fig. 22.6) calls `ClipPlayer` method `play` (lines 109–112), the `Clip` method `start` begins audio playback.

Class `ClipPlayerTest` (Fig. 22.6) enables users to specify an audio file to play by clicking the `Open Audio Clip` button. When users click the button, method `actionPerformed` (lines 37–58) prompts an audio file name and location (line 39) and creates
a ClipPlayer for the specified audio file (line 44). Line 47 invokes ClipPlayer method openFile, which returns true if the ClipPlayer can open the audio file. If so, line 50 calls ClipPlayer method play to play the audio and line 53 calls ClipPlayer method setReplay to indicate that the audio should not loop continuously.

Performance Tip 22.4

Large audio files take a long time to load, depending on the speed of the computer. An alternative playback form is to buffer the audio by loading a portion of the data to begin playback and continuing to load the remainder as the audio plays. This is similar to the streaming capability provided by JMF.

```
// Fig. 22.6: ClipPlayerTest.java
// Test file for ClipPlayer

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

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

public class ClipPlayerTest extends JFrame {

    // object to play audio clips
    private ClipPlayer clipPlayer;

    // constructor for ClipPlayerTest
    public ClipPlayerTest()
    {
        super( "Clip Player" );

        // panel containing buttons
        JPanel buttonPanel = new JPanel();
        getContentPane().add( buttonPanel );

        // open file button
        JButton openFile = new JButton( "Open Audio Clip" );
        buttonPanel.add( openFile, BorderLayout.CENTER );

        // register ActionListener for openFile events
        openFile.addActionListener(

            new ActionListener() {

                public void actionPerformed( ActionEvent event )
                {
                    File mediaFile = getFile();

                    // try to open and play an audio clip file
                    public void actionPerfomed( ActionEvent event )
                    {
                        File mediaFile = getFile();
```
if ( mediaFile != null ) {
    // instantiate new clip player with mediaFile
    clipPlayer = new ClipPlayer( mediaFile );

    // if clip player opened correctly
    if ( clipPlayer.openFile() == true ) {
        // play loaded clip
        clipPlayer.play();

        // no replay
        clipPlayer.setReplay( false );
    }
}

// execute application
public static void main( String args[] )
{
    ClipPlayerTest test = new ClipPlayerTest();

    test.setSize( 150, 70 );
test setLocation( 300, 300 );
test setDefaultCloseOperation( EXIT_ON_CLOSE );
test setVisible( true );
}
22.7 Musical Instrument Digital Interface (MIDI)

The Musical Instrument Digital Interface (MIDI) is a standard format for electronic music. MIDI music can be created through a digital instrument, such as an electronic keyboard, or through software. The MIDI interface allows musicians to create synthesized digital music that reproduces the actual music. Then they can share their musical creations with music enthusiasts around the world. A MIDI synthesizer is a device that can produce MIDI sounds and music.

Programs can easily manipulate MIDI data. Like other types of audio, MIDI data has a well-defined format that MIDI players can interpret, play and use to create new MIDI data. The Complete Detailed MIDI 1.0 specification provides detailed information on MIDI files. Visit the official MIDI Web site at www.midi.org for information on MIDI and its specification. Java Sound’s MIDI packages (javax.sound.midi and javax.sound.midi.spi) allow developers to access MIDI data.

Interpretation of MIDI data varies between synthesizers, so a file may sound quite different when played on synthesizers other than the one on which it was created. Synthesizers support varying types and numbers of instrumental sounds and different numbers of simultaneous sounds. Usually hardware-based synthesizers are capable of producing higher-quality synthesized music than software-based synthesizers.

Many Web sites and games use MIDI for music playback, as it enables developers to entertain users with lengthy, digitized music files that do not require a lot of memory. In comparison, sampled audio files can grow to be quite large. Package javax.sound.midi enables programs to manipulate, play and synthesize MIDI. Java Sound supports MIDI files with mid and rmf (Rich Music Format or RMF) extensions.

The example presented in Sections 22.7.1 through 22.7.4 covers MIDI synthesis, playback, recording and saving. Class MidiDemo (Fig. 22.10) is the main application class that utilizes classes MidiData (Fig. 22.7), MidiRecord (Fig. 22.8) and MidiSynthesizer (Fig. 22.9). Class MidiSynthesizer provides resources for generating sounds...
and transmitting them to other MIDI devices, such as recorders. Class MidiData handles MIDI playback, track initialization and event information. Class MidiRecord provides MIDI recording capabilities. Class MidiDemo ties the other classes together with an interactive GUI that includes a simulated piano keyboard, play and record buttons, and a control panel for configuring MIDI options. Class MidiDemo also uses MIDI event-synchronization to play a MIDI file and highlight the appropriate piano keys, simulating someone playing the keyboard.

An integral part of this MIDI example is its GUI, which allows users to play musical notes on a simulated piano keyboard (see screen capture in Fig. 22.10). When the mouse hovers over a piano key, the program plays the corresponding note. In this section, we refer to this as user synthesis. The Play MIDI button in the GUI allows the user to select a MIDI file to play. The Record button records the notes played on the piano (user synthesis). Users can save the recorded MIDI to a file using the Save MIDI button and play back the recorded MIDI using the PlayBack button. Users can click the Piano Player button to open a MIDI file, then play that file back through a synthesizer. The program signifies synchronization of notes and piano keys by highlighting the key that corresponds to the note number. This playback and synchronization ability is called the “piano player.” While the “piano player” is running, users can synthesize additional notes, and record both the old audio material and the new user-synthesized notes by clicking the Record button. The JComboBox in the upper-left corner of the GUI enables users to select an instrument for synthesis. Additional GUI components include a volume control for user-synthesized notes and a tempo control for controlling the speed of the “piano player.”

Testing and Debugging Tip 22.1

Testing the MIDI file playback functions requires a sound card and an audio file in MIDI format.

22.7.1 MIDI Playback

This section discusses how to play MIDI files and how to access and interpret MIDI file contents. Class MidiData (Fig. 22.7) contains methods that load a MIDI file for playback. The class also provides the MIDI track information required by the “piano player” feature. A MIDI sequencer is used to play and manipulate the audio data. Often, MIDI data is referred to as a sequence, because the musical data in a MIDI file is composed of a sequence of events. The steps performed in MIDI playback are accessing a sequencer, loading a MIDI sequence or file into that sequencer and starting the sequencer.
public class MidiData {

    // MIDI track data
    private Track track;

    // player for MIDI sequences
    private Sequencer sequencer;

    // MIDI sequence
    private Sequence sequence;

    // MIDI events containing time and MidiMessages
    private MidiEvent currentEvent, nextEvent;

    // MIDI message usually containing sounding messages
    private ShortMessage noteMessage;

    // short, meta, or sysex MIDI messages
    private MidiMessage message;

    // index of MIDI event in track, command in MIDI message
    private int eventIndex = 0, command;

    // method to play MIDI sequence via sequencer
    public void play() {
        // initiate default sequencer
        try {
            // get sequencer from MidiSystem
            sequencer = MidiSystem.getSequencer();

            // open sequencer resources
            sequencer.open();

            // load MIDI into sequencer
            sequencer.setSequence(sequence);

            // play sequence
            sequencer.start();
        } catch (MidiUnavailableException noMidiException) {
            noMidiException.printStackTrace();
        } catch (InvalidMidiDataException badMidiException) {
            badMidiException.printStackTrace();
        }
    }

Fig. 22.7 MidiData loads MIDI files for playback (part 2 of 5).
// end method play

// method returning adjusted tempo/resolution of MIDI
public int getResolution()
{
    return 500 / sequence.getResolution();
}

// obtain MIDI and prepare track in MIDI to be accessed
public boolean initialize( File file )
{
    // get valid MIDI from file into sequence
    try {
        sequence = MidiSystem.getSequence( file );
    }
    
    // unreadable MIDI file or unsupported MIDI
    catch ( InvalidMidiDataException badMIDI ) {
        badMIDI.printStackTrace();
        return false;
    }
    
    // I/O error generated during file reading
    catch ( IOException ioException ) {
        ioException.printStackTrace();
        return false;
    }
    
    return true;
}

// prepare longest track to be read and get first MIDI event
public boolean initializeTrack()
{
    // get all tracks from sequence
    Track tracks[] = sequence.getTracks();
    
    if ( tracks.length == 0 ) {
        System.err.println( "No tracks in MIDI sequence!" );
        return false;
    }
    
    track = tracks[ 0 ];
    
    // find longest track
    for ( int i = 0; i < tracks.length; i++ )
    {
        if ( tracks[ i ].size() > track.size() )
            track = tracks[ i ];

Fig. 22.7 MidiData loads MIDI files for playback (part 3 of 5).
// set current MIDI event to first event in track
    currentEvent = track.get( eventIndex );

// get MIDI message from event
    message = currentEvent.getMessage();

// track initialization successful
    return true;
}

// move to next event in track
public void goNextEvent()
{
    eventIndex++;
    currentEvent = track.get( eventIndex );
    message = currentEvent.getMessage();
}

// get time interval between events
public int getEventDelay()
{
    // first event's time interval is its duration
    if ( eventIndex == 0 )
        return ( int ) currentEvent.getTick();

    // time difference between current and next event
    return ( int ) ( track.get( eventIndex + 1 ).getTick() -
                    currentEvent.getTick() );
}

// return if track has ended
public boolean isTrackEnd()
{
    // if eventIndex is less than track's number of events
    if ( eventIndex + 1 < track.size() )
        return false;

    return true;
}

// get current ShortMessage command from event
public int getEventCommand()
{
    if ( message instanceof ShortMessage ) {

        // obtain MidiMessage for accessing purposes
        noteMessage = ( ShortMessage ) message;
        return noteMessage.getCommand();
    }

    return -1;
}

Fig. 22.7  MidiData loads MIDI files for playback (part 4 of 5).
To play a MIDI file with a sequence, a program must obtain the MIDI sequence and check for compatibility issues. `MidiData` method `initialize` (lines 73–94) obtains a `Sequence` of MIDI data from a file with `MidiSystem` method `getSequence` (line 77). A `Sequence` contains MIDI tracks, which, in turn, contain MIDI events. Each event encapsulates a MIDI message of instructions for the MIDI devices. Individual tracks of a MIDI sequence are analogous to tracks on a CD. However, while CD tracks are played in order, MIDI tracks are played in parallel. A MIDI track is a recorded sequence of data. MIDI's usually contain multiple tracks. Method `getSequence` can also obtain a MIDI sequence from a URL or an `InputStream`. Method `getSequence` throws an `InvalidMidiDataException` if the MIDI system detects an incompatible MIDI file.

**Portability Tip 22.2**

Because of incompatible file parsers in different operating systems, sequencers may not be able to play RMF files.

After obtaining a valid MIDI sequence, the program must obtain a sequencer and load the sequence into the sequencer. Method `play` (lines 35–64) in class `MidiData` calls `MidiSystem` method `getSequencer` (line 41) to obtain a `Sequencer` to play the `Sequence`. Interface `Sequencer`, which extends interface `MidiDevice` (the super-interface for all MIDI devices), provides the standard sequencer device to play MIDI data. If another program is using the same `Sequencer` object, method `getSequencer` throws a `MidiUnavailableException`. Line 44 calls `Sequencer` method `open` to prepare to play a `Sequence`. `Sequencer` method `setSequence` (line 47) loads a MIDI `Sequence` into the `Sequencer` and throws an `InvalidMidiException` if the `Sequencer` detects an unrecognizable MIDI sequence. Line 50 begins playing the MIDI sequence by calling the `Sequencer`'s `start` method.

In addition to MIDI playback methods, class `MidiData` also provides methods that enable a program to access the events and messages of a MIDI sequence. As we shall see, class `MidiDemo` (Figure 22.10) uses class `MidiData` to access the data in a MIDI file for synchronizing the highlighting of piano keys. The MIDI events are stored in the MIDI’s
tracks, which are instances of class **Track** (package `javax.sound.midi`). MIDI events in MIDI tracks are represented by class **MidiEvent** (package `javax.sound.midi`). Each MIDI event contains an instruction and the time at which it should occur. The individual events in a track contain messages of type **MidiMessage** that specify the MIDI instructions for a **MidiDevice**. There are three types of MIDI messages—**ShortMessage**, **SysexMessage** and **MetaMessage**. **ShortMessages** are explicit musical instructions, such as the specific notes to play and pitch changes. The other two less-used messages are **SysexMessages**, system-exclusive messages for MIDI devices, and **MetaMessages**, which may indicate to a MIDI device that the MIDI has reached the end of a track. This section deals exclusively with **ShortMessages** that play specific notes.

Next, the program must obtain the tracks and read their events. **MidiData** method `initializeTrack` (lines 97–125) invokes `Sequence`’s `getTracks` method (line 100) to obtain all of the tracks in the MIDI sequence. Lines 108–114 determine the longest track in the MIDI and set it as the one to play. Line 117 obtains the first MIDI event in the **Track** by invoking its `get` method with the index of the event in the track as the parameter. At this point `eventIndex` is set to 0 (line 32). Line 120 obtains the MIDI message from the MIDI event using method `getMessage` of class **MidiEvent**. To help a program step through each event in the tracks, the program can call **MidiData** method `goNextEvent` (lines 128–133) to load the next event and message. Method `goNextEvent` increments `eventIndex` in the loaded MIDI **Track** and finds the next event’s **MidiMessage**.

In addition to reading the events, the program must also determine how long each event lasts and the spacing between events. Method `getEventDelay` (lines 136–145) returns the duration of a **MidiEvent** as the time difference between two events in the MIDI sequence (lines 143–144). The **MidiEvent**’s `getTick` method provides the specific time when the event takes place (also called a *time stamp*). Lines 139–140 return the first **MidiEvent**’s time stamp as the event’s duration.

**Class MidiData** provides other methods to return the commands, the note numbers and the volume of note-related **ShortMessages**. Method `getEventCommand` (lines 158–168) determines the command number representing the command instruction. Line 160 of method `getEventCommand` indicates whether the currently loaded **MidiMessage** is a **ShortMessage**. If so, line 163 assigns the **ShortMessage** to object `noteMessage` and line 164 returns the **ShortMessage**’s command status byte by invoking **ShortMessage**’s `getCommand` method. Method `getEventCommand` returns -1 if the event does not contain a **ShortMessage**. **MidiData** method `getNote` (lines 171–177) invokes **ShortMessage** method `getData1` (line 174) to return the note number. Method `getVolume` (lines 180–183) invokes **ShortMessage** method `getData2` to return the volume. **Class MidiData** also provides an indication of the end of a track in method `isTrackEnd` (lines 148–155), which determines whether the event index has surpassed the number of events in the track (line 151).

### 22.7.2 MIDI Recording

A program can record MIDI using a sequencer. **Class MidiRecord** (Fig. 22.8) handles the recording functions of this MIDI demo using an object that implements interface **Sequencer** as a MIDI recorder. As long as the MIDI devices are set up correctly, interface **Sequencer** provides simple methods for recording. **Class MidiRecord** has a constructor (lines 29–32) that receives an object that implements interface **Transmitter** as an
argument. A **Transmitter** sends MIDI messages to a MIDI device that implements interface **Receiver**. Think of **Transmitters** and **Receivers** as output and input ports respectively for MIDI devices.

```java
// Fig. 22.8: MidiRecord.java
// Allows for recording and playback
// of synthesized MIDI

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

// Java extension package
import javax.sound.midi.*;

public class MidiRecord {

    // MIDI track
    private Track track;

    // MIDI sequencer to play and access music
    private Sequencer sequencer;

    // MIDI sequence
    private Sequence sequence;

    // receiver of MIDI events
    private Receiver receiver;

    // transmitter for transmitting MIDI messages
    private Transmitter transmitter;

    public MidiRecord( Transmitter transmit )
    {
        transmitter = transmit;
    }

    // initialize recording sequencer, set up recording sequence
    public boolean initialize()
    {
        try {
            // create empty MIDI sequence and set up sequencer wiring
            sequence = new Sequence( Sequence.PPQ, 10 );

            // obtain sequencer and open it
            sequencer = MidiSystem.getSequencer();
            sequencer.open();

            // get receiver of sequencer
            receiver = sequencer.getReceiver();
        }
    }

    //...remaining code...
}
```

**Fig. 22.8** **MidiRecord** enables a program to record a MIDI sequence (part 1 of 3).
if ( receiver == null ) {
    System.err.println("Receiver unavailable for sequencer");
    return false;
}

// set receiver for transmitter to send MidiMessages
transmitter.setReceiver( receiver );

makeTrack();

// invalid timing division specification for new sequence
catch ( InvalidMidiDataException invalidMidiException ) {
    invalidMidiException.printStackTrace();
    return false;
}

// sequencer or receiver unavailable
catch ( MidiUnavailableException noMidiException ) {
    noMidiException.printStackTrace();
    return false;
}

// MIDI recorder initialization successful
return true;

} // end method initialize

// make new empty track for sequence
public void makeTrack()
{
    // if previous track exists, delete it first
    if ( track != null )
        sequence.deleteTrack( track );

    // create track in sequence
    track = sequence.createTrack();
}

// start playback of loaded sequence
public void play()
{
    sequencer.start();
}

// start recording into sequence
public void startRecord()
{
    // load sequence into recorder and start recording
    try {
        sequencer.setSequence( sequence );
    }

Fig. 22.8  MidiRecord enables a program to record a MIDI sequence (part 2 of 3).
The first step of recording MIDI data is similar to the playback mechanism in class 
MidiData. In addition to obtaining an empty sequence and a sequencer, a MIDI recording 
program needs to connect the transmitters and receivers. After successfully “wiring” the 
sequencer’s receiver as the “IN PORT,” the recorder loads the empty sequence into the 
sequencer to start recording to a new track in the sequence. The following discussion covers 
these steps.
Method `initialize` (lines 35–77) of class `MidiRecord` sets up the sequencer for recording. Line 41 of method `initialize` instantiates an empty sequence. `MidiRecord` will record data to the empty sequence once the transmitter is connected to the receiver. Line 48 obtains the recording sequencer’s `receiver` and line 57 specifies that `transmitter` will send its messages to `receiver`.

MIDI messages must be placed in a track, so method `initialize` invokes method `makeTrack` (lines 80–88) to delete the previous existing `track` (line 84) and to create an empty `Track` (line 87). Method `makeTrack` can also be called from an external class to record a new sequence without instantiating new sequencers and a new sequence.

After setting up a sequencer and an empty sequence, calling `MidiRecord` method `startRecord` (lines 97–115) starts the recording process. Line 101 loads an empty sequence into the sequencer. `Sequencer` method `recordEnable` is called and passed the `track` object and a channel number as arguments (line 104), which enables recording on that `track`. Line 106 invokes `Sequencer`’s `startRecording` method to start the recording of MIDI events sent from the transmitter. `Sequencer`’s `stopRecording` method stops recording and is called in `MidiRecord`’s `stopRecord` method (lines 118–121).

Class `MidiRecord` can also supports save a recorded sequence to a MIDI file using its `saveSequence` method (lines 124–144). Although most MIDI sequences can support `MIDI type 0 files` (the most common type of MIDI file), the sequence should be checked for other supported file types. Line 127 obtains an array of MIDI file types supported by the system for writing a sequence to a file. The MIDI file types are represented by integer values of 0, 1 or 2. Using the first supported file type, the `MidiSystem` writes the sequence to a specified `File` (line 136) passed into method `saveSequence` as an argument. `MidiRecord`’s `play` method (lines 91–94) enables the program to play back the newly recorded sequence.

### 22.7.3 MIDI Synthesis

This `MidiDemo` program provides an interactive piano that generates notes according to the keys pressed by the user. Class `MidiSynthesizer` (Fig. 22.9) generates these notes directly, and sends them to another device. Specifically, it sends the notes to a sequencer’s `receiver` through a `transmitter` to record the MIDI sequence. Class `MidiSynthesizer` uses an object that implements interface `Synthesizer` (a sub-interface of `MidiDevice`) to access the default synthesizer’s sound generation, instruments, channel resources and sound banks. A `SoundBank` is the container for various `Instruments`, which instructs the computer on how to make the sound of a specific note. Different notes made by various instruments are played through a `MidiChannel` on different tracks simultaneously to produce symphonic melodies.

```
1 // Fig. 22.9: MidiSynthesizer.java
2 // Accessing synthesizer resources
3
4 // Java extension package
5 import javax.sound.midi.*;
6
```

*Fig. 22.9*  `MidiSynthesizer` can generate notes and send them to another MIDI device (part 1 of 4).
public class MidiSynthesizer {

    // main synthesizer accesses resources
    private Synthesizer synthesizer;

    // available instruments for synthesis use
    private Instrument instruments[];

    // channels through which notes sound
    private MidiChannel channels[];
    private MidiChannel channel;    // current channel

    // transmitter for transmitting messages
    private Transmitter transmitter;

    // receiver end of messages
    private Receiver receiver;

    // short message containing sound commands, note, volume
    private ShortMessage message;

    // constructor for MidiSynthesizer
    public MidiSynthesizer()
    {
        // open synthesizer, set receiver,
        // obtain channels and instruments
        try {
            synthesizer = MidiSystem.getSynthesizer();
            if ( synthesizer != null ) {
                synthesizer.open();
                transmitter = synthesizer.getTransmitter();
                if ( transmitter == null )
                    System.err.println( "Transmitter unavailable" );
                receiver = synthesizer.getReceiver();
                if ( receiver == null )
                    System.out.println( "Receiver unavailable" );
                instruments = synthesizer.getAvailableInstruments();
                channels = synthesizer.getChannels();
            }
        }
    }

Fig. 22.9 MidiSynthesizer can generate notes and send them to another MIDI device (part 2 of 4).
// assign first channel as default channel
channel = channels[ 0 ];

// synthesizer, receiver or transmitter unavailable
catch ( MidiUnavailableException noMidiException ) {
   noMidiException.printStackTrace();
}

// return available instruments
public Instrument[] getInstruments()
{
   return instruments;
}

// return synthesizer's transmitter
public Transmitter getTransmitter()
{
   return transmitter;
}

// sound note on through channel
public void midiNoteOn( int note, int volume )
{
   channel.noteOn( note, volume );
}

// sound note off through channel
public void midiNoteOff( int note )
{
   channel.noteOff( note );
}

// change to selected instrument
public void changeInstrument( int index )
{
   Patch patch = instruments[ index ].getPatch();
   channel.programChange( patch.getBank(),
                         patch.getProgram() );
}

// send custom MIDI messages through transmitter
public void sendMessage( int command, int note, int volume )
{

Fig. 22.9  MidiSynthesizer can generate notes and send them to another MIDI device (part 3 of 4).
MidiSynthesizer’s constructor (lines 29–72) acquires the synthesizer and initializes related resources. Line 34 obtains a Synthesizer object from the MidiSystem and line 38 opens the Synthesizer. To enable sounds to be played and recorded at the same time, lines 41–47 obtain the Transmitter and Receiver of the Synthesizer. When a MIDI message is sent to the synthesizer’s receiver, the synthesizer executes the message’s instruction, generating notes, and the transmitter sends that message to designated Receivers of other MidiDevices.

**Common Programming Error 22.3**

A MidiUnavailableException occurs when a program attempts to acquire unavailable MidiDevice resources such as synthesizers and transmitters.

MIDI messages are sent to the MidiSynthesizer from MidiDemo as a result of either pressing a piano key or a MidiEvent in the preloaded track of MidiData. A note can be generated by accessing the channels of the synthesizer directly. For simplicity, MidiSynthesizer uses only the first channel (out of a possible 16) to sound notes. Line 57 invokes Synthesizer method getChannels to obtain all 16 channels from synthesizer, and line 60 sets the default channel to the first channel. A MidiChannel sounds a note by calling its noteOn method with the note number (0–127) and a volume number as arguments. MidiChannel’s noteOff method turns off a note with just the note number as an argument. MidiSynthesizer accesses these MidiChannel methods through method midiNoteOn (lines 87–90) and method midiNoteOff (lines 93–96), respectively.

A synthesizer can use its default instruments to sound notes. Line 54 obtains the default instrument available through the synthesizer or through a default sound bank by invoking Synthesizer method getAvailableInstruments. A sound bank usually has 128 instruments. The instrument in use can be changed by invoking MidiSynthesizer
method `changeInstrument` (lines 99–105). Lines 103–104 invoke `MidiChannel`’s `programChange` method to load the desired instrument program with the bank and program number obtained from `patch` (line 104) as the parameters. A `Patch` is the location of a loaded instrument.

**Performance Tip 22.5**

A program can import more instruments by loading a customized sound bank through `Synthesizer` method `loadAllInstruments` with a `SoundBank` object.

By sending `MidiMessages` to a `Synthesizer`’s `Receiver`, a program can invoke the synthesizer to sound notes without using its channels. Sending `MidiMessages` to a `MidiDevice`’s `Receiver` also allows the device’s `Transmitter`s to send these messages to another `MidiDevice`’s `Receiver`.

In `MidiSynthesizer`’s `sendMessage` method (lines 108–127), lines 112–116 create a new `ShortMessage` from the parameters of method `sendMessage` and send the message to the synthesizer’s receiver (line 119). Line 116 of method `sendMessage` invokes `ShortMessage` method `setMessage` to set the contents of the message’s instructions using three `int` arguments: a command, the note to play and the volume of the note. Method `setMessage` throws an `InvalidMidiDataException` if the designated command and parameter values are invalid.

When creating a new `ShortMessage` using method `setMessage`, the meaning of the second and third arguments vary depending on the command. Command `ShortMessage.NOTE_ON` designates the second parameter to be the note number and third argument to be the velocity (i.e. volume) of the note. The `ShortMessage.PROGRAM_CHANGE` command designates the second argument as the instrument program to use and ignores the third argument.

Line 119 sends the created `ShortMessage` to the synthesizer’s `receiver` by calling `Receiver` method `send` with the `MidiMessage` and a time stamp as its arguments. `MidiSynthesizer` does not deal with the complexity of MIDI synthesis timing. The receiver sends a value of `-1` for the time stamp parameter to designate that the time stamp should be ignored. The sequence recorder in class `MidiRecord` takes care of timing issues when it receives the messages.

Up to this point, we have discussed the tools needed to create our MIDI piano. In brief synopsis, class `MidiDemo` (Fig. 22.10) uses class `MidiSynthesizer` to generate sounds and to access channels and instruments. `MidiDemo` uses `MidiData` to playback MIDI files and access MIDI track information. `MidiRecord` provides the recording function for `MidiDemo`, which receives messages from `MidiSynthesizer`.

### 22.7.4 Class MidiDemo

We now present class `MidiDemo` (Fig. 22.10), which provides the GUI for our piano as well as other GUI components to control the capabilities of this example.

Using a `for` loop, utility method `makeKeys` (lines 86–155) in class `MidiDemo` creates 64 buttons that represent 64 different piano keys. Whenever the mouse hovers over a key, the program sounds the designated note. Method `makeKeys` arranges the keys at the bottom of the frame using each button’s `setBounds` method (line 106) to designate the location and size of the buttons. The program arranges the buttons horizontally according to their index in array `noteButton`.
Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 1 of 14).
// constructor for MidiDemo
public MidiDemo()
{
    super( "MIDI Demo" );

    container = getContentPane();
    container.setLayout( new BorderLayout() );

    // synthesizer must be instantiated to enable synthesis
    midiSynthesizer = new MidiSynthesizer();

    // make piano keys
    makeKeys();

    // add control panel to frame
    controlPanel = new JPanel( new BorderLayout() );
    container.add( controlPanel, BorderLayout.NORTH );

    makeConfigureControls();

    // add button panel to frame
    buttonPanel = new JPanel( new GridLayout( 5, 1 ) );
    controlPanel.add( buttonPanel, BorderLayout.EAST );

    // make GUI
    makePlaySaveButtons();
    makeRecordButton();
    makePianoPlayerButton();

}  // end constructor

// utility method making piano keys
private void makeKeys()
{
    // panel containing keys
    JPanel keyPanel = new JPanel( null );
    container.add( keyPanel, BorderLayout.CENTER );

    // piano keys
    noteButton = new JButton[ MAX_KEYS ];

    // add MAX_KEYS buttons and what note they sound
    for ( int i = 0; i < MAX_KEYS; i++ ) {
        final int note = i;
        noteButton[ i ] = new JButton();

        // setting white keys
        noteButton[ i ].setBackground( Color.white );
}

Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 2 of 14).
// set correct spacing for buttons
noteButton[i].setBounds( i * 11, 1, 11, 40);
keyPanel.add( noteButton[i] );

// register a mouse listener for mouse events
noteButton[i].addMouseListener(
    new MouseAdapter() {

        // invoke key note when mouse touches key
        public void mouseEntered( MouseEvent mouseEvent )
        {
            // if recording, send message to receiver
            if ( recording )
                midiSynthesizer.sendMessage( ShortMessage.NOTE_ON, note + FIRST_NOTE, midiVolume );

            // else just sound the note
            else
                midiSynthesizer.midiNoteOn( note + FIRST_NOTE, midiVolume );

            // turn key color to blue
            noteButton[ note ].setBackground( Color.blue );
        }

        // turn key note off when mouse leaves key
        public void mouseExited( MouseEvent mouseEvent )
        {
            if ( recording )
                midiSynthesizer.sendMessage( ShortMessage.NOTE_OFF, note + FIRST_NOTE, midiVolume );

            else
                midiSynthesizer.midiNoteOff( note + FIRST_NOTE );

            noteButton[ note ].setBackground( Color.white );
        }
    }  // end MouseAdapter
); // end call to addMouseListener

} // end method makeKeys

Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 3 of 14).
private void makeConfigureControls()
{
    JPanel configurePanel =
            new JPanel( new GridLayout( 5, 1 ) );
    controlPanel.add( configurePanel, BorderLayout.WEST );
    instrumentBox = new JComboBox(
            midiSynthesizer.getInstruments() );
    configurePanel.add( instrumentBox );

    // register an ActionListener for instrumentBox events
    instrumentBox.addActionListener(
            new ActionListener() {

                    // change current instrument program
                    public void actionPerformed( ActionEvent event )
                    {
                        // change instrument in synthesizer
                        midiSynthesizer.changeInstrument( instrumentBox.getSelectedItem() );
                    }
            });

    JLabel volumeLabel = new JLabel( "volume" );
    configurePanel.add( volumeLabel );

    volumeSlider = new JSlider( SwingConstants.HORIZONTAL, 5, 80, 30 );

    // register a ChangeListener for slider change events
    volumeSlider.addChangeListener(
            new ChangeListener() {

                    // change volume
                    public void stateChanged( ChangeEvent changeEvent )
                    {
                        midiVolume = volumeSlider.getValue();
                    }
            });
}

Fig. 22.10  MidiDemo provides the GUI than enables users to interact with the application (part 4 of 14).
configurePanel.add( volumeSlider );

JLabel tempLabel = new JLabel( "tempo" );
configurePanel.add( tempLabel );

resolutionSlider = new JSlider( SwingConstants.HORIZONTAL, 1, 10, 1 );

// register a ChangeListener slider for change events
resolutionSlider.addChangeListener( new ChangeListener() {

    // change resolution if value changed
    public void stateChanged( ChangeEvent changeEvent ) {
        resolution = resolutionSlider.getValue();
    }

} // end ChangeListener
);

resolutionSlider.setEnabled( false );
configurePanel.add( resolutionSlider );

} // end method makeConfigureControls

// set up play and save buttons
private void makePlaySaveButtons() {

    playButton = new JButton( "Playback" );

    // register an ActionListener for playButton events
    playButton.addActionListener( new ActionListener() {

    // anonymous inner class to handle playButton event
    new ActionListener() {

        // playback last recorded MIDI
        public void actionPerformed( ActionEvent event ) {
            if ( midiRecord != null )
                midiRecord.play();
        }

    } // end ActionListener

}); // end call to method addActionListener

Fig. 22.10  MidiDemo provides the GUI than enables users to interact with the application (part 5 of 14).
buttonPanel.add( playButton );
playButton.setEnabled( false );

listenButton = new JButton( "Play MIDI" );
// register an ActionListener for listenButton events
listenButton.addActionListener(

    // anonymous inner class to handle listenButton events
    new ActionListener() {

        // playback MIDI file
        public void actionPerformed( ActionEvent event ) {
            File midiFile = getFile();

            if ( midiFile == null )
                return;

            midiData = new MidiData();

            // prepare MIDI track
            if ( midiData.initialize( midiFile ) == false )
                return;

            // play MIDI data
            midiData.play();
        }
    } // end ActionListener
);

// end call to method addActionListener
buttonPanel.add( listenButton );

saveButton = new JButton( "Save MIDI" );
// register an ActionListener for saveButton events
saveButton.addActionListener(

    // anonymous inner class to handle saveButton events
    new ActionListener() {

        // get save file and save recorded MIDI
        public void actionPerformed( ActionEvent event ) {
            File saveFile = getSaveFile();

            if ( saveFile != null )
                midiRecord.saveSequence( saveFile );
        }
    } // end ActionListener
);
Fig. 22.10  MidiDemo provides the GUI than enables users to interact with the application (part 7 of 14).
364     // stop recording when button is "stop" button
365     else {
366         midiRecord.stopRecord();
367
368         recordButton.setText( "Record" );
369         recording = false;
370
371         playButton.setEnabled( true );
372         saveButton.setEnabled( true );
373     }
374
375 }  // end method actionPerformed
376
377 }  // end ActionListener
378
379 }  // end call to method addActionListener
380
381         buttonPanel.add( recordButton );
382
383 } // end method makeRecordButton
384
385 // create Piano Player button and functionality
386 private void makePianoPlayerButton()
387 {
388     pianoPlayerButton = new JButton( "Piano Player" );
389
390     // register an ActionListener for pianoPlayerButton events
391     pianoPlayerButton.addActionListener(
392
393         // anonymous inner class to handle pianoPlayerButton
394         new ActionListener() {
395
396             // initialize MIDI data and piano player timer
397             public void actionPerformed( ActionEvent event )
398             {
399                 File midiFile = getFile();
400
401                 if ( midiFile == null )
402                     return;
403
404                 midiData = new MidiData();
405
406                 // prepare MIDI track
407                 if ( midiData.initialize( midiFile ) == false )
408                     return;
409
410                 if ( midiData.initializeTrack() == false )
411                     return;
412
413                 // set initial resolution from MIDI
414                 resolution = midiData.getResolution();
415
Fig. 22.10  MidiDemo provides the GUI than enables users to interact with the application (part 8 of 14).
// new instance of timer for handling
// piano sounds and key pressing with tempo
pianoTimer = new Timer(
    midiData.getEventDelay() * resolution,
    new TimerHandler() );
l paranButton.setEnabled( false );
pianoPlayerButton.setEnabled( false );
resolutionSlider.setEnabled( true );
pianoTimer.start();
}

}  // end actionPerformed
}  // end ActionListener
);

buttonPanel.add( pianoPlayerButton );

}  // end method makePianoPlayerButton

// inner class handles MIDI timed events
private class TimerHandler implements ActionListener {

    // simulate key note of event if present, jump to next
    // event in track and set next delay interval of timer
    // method invoked when timer reaches next event time
public void actionPerformed( ActionEvent actionEvent )
{
    // if valid last key on, set it white
    if ( lastKeyOn != -1 )
        noteButton[ lastKeyOn ].setBackground( Color.white );

    noteAction();
    midiData.goNextEvent();

    // stop piano player when end of MIDI track
    if ( midiData.isTrackEnd() == true ) {
        if ( lastKeyOn != -1 )
            noteButton[ lastKeyOn ].setBackground( Color.white );

        pianoTimer.stop();
        listenButton.setEnabled( true );
pianoPlayerButton.setEnabled( true );
resolutionSlider.setEnabled( false );
    return;

Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 9 of 14).
Chapter 22   Java Media Framework and Java Sound (on CD)   1309

// set interval before next sounding event
pianoTimer.setDelay( midiData.getEventDelay() * resolution );

} // end actionPerformed method

} // end inner class TimerHandler

// determine which note to sound
// according to MIDI messages
private void noteAction()
{
  // during Note On message, sound note and press key
  if ( midiData.getEventCommand() ==
       ShortMessage.NOTE_ON ) {

    // make sure valid note is in range of keys
    if ( ( midiData.getNote() >= FIRST_NOTE ) &&
         ( midiData.getNote() < FIRST_NOTE + MAX_KEYS ) ) {

      lastKeyOn = midiData.getNote() - FIRST_NOTE;
      // set key color to red
      noteButton[ lastKeyOn ].setBackground( Color.red );
      // send and sound note through synthesizer
      midiSynthesizer.sendMessage( 144,
                                   midiData.getNote(), midiData.getVolume() );

    } // end if

  } // else no last key pressed
  else
    lastKeyOn = -1;

} // end if

// receiving Note Off message will sound off note
// and change key color back to white
else

  // if message command is note off
  if ( midiData.getEventCommand() ==
       ShortMessage.NOTE_OFF ) {

    if ( ( midiData.getNote() >= FIRST_NOTE ) &&
         ( midiData.getNote() < FIRST_NOTE + MAX_KEYS ) ) {

    } // end if

Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 10 of 14).
// set appropriate key to white
noteButton[ midiData.getNote() - FIRST_NOTE ].setBackground( Color.white );

// send note off message to receiver
midiSynthesizer.sendMessage( 128, midiData.getNote(), midiData.getVolume() );

} // end if

} // end method noteAction

public File getSaveFile()
{
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
    int result = fileChooser.showSaveDialog( this );
    if ( result == JFileChooser.CANCEL_OPTION )
        return null;
    else
        return fileChooser.getSelectedFile();
}

// get file from computer
public File getFile()
{
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
    int result = fileChooser.showOpenDialog( this );
    if ( result == JFileChooser.CANCEL_OPTION )
        return null;
    else
        return fileChooser.getSelectedFile();
}

// execute application
public static void main( String args[] )
{
    MidiDemo midiTest = new MidiDemo();
    midiTest.setSize( 711, 225 );
    midiTest.setDefaultCloseOperation ( EXIT_ON_CLOSE );

Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 11 of 14).
571    midiTest.setVisible( true );
572  }
573
574 } // end class MidiDemo

Fig. 22.10 MidiDemo provides the GUI than enables users to interact with the application (part 12 of 14).
Fig. 22.10 **MidiDemo** provides the GUI that enables users to interact with the application (part 13 of 14).
Fig. 22.10  **MidiDemo** provides the GUI that enables users to interact with the application (part 14 of 14).
Look-and-Feel Observation 22.3

To arrange GUI components at specific locations without the help of Layout Managers, set the layout of the panel containing the components to null. By default, a JPanel sets a FlowLayout LayoutManager when the panel is instantiated with no arguments.

Lines 110–151 register MouseListeners for each piano-key button. The program calls method mouseEntered (lines 116–132) when the mouse hovers over that button. If the program is not in recording mode, method mouseEntered directly accesses the channels in class MidiSynthesizer to sound the note (lines 125–127). Otherwise, method mouseEntered invokes MidiSynthesizer’s sendMessage method to send a note message to the synthesizer and to the recording device (lines 119–122). Lines 130–131 set the button’s background color to blue to indicate that the note is being played. When the mouse is no longer hovering over the button, the program calls method mouseExited (lines 135–147) to turn off the note and change the button’s background to its original color.

Out of the possible 128 notes, only the middle 64 notes are accessible via the piano in the example. The range of notes can be changed by modifying the constants in line 52. Constant FIRST_NOTE is the value of the first key and the sum of FIRST_NOTE and MAX_KEYS is the value of the last key. Constant MAX_KEYS specifies the number of piano keys to create.

Class MidiDemo invokes method makeConfigureControls (lines 158–237) to setup the program’s MIDI controls, which consist of an instrument selector JComboBox, a user-synthesis volume changer JSlider and a “piano player” tempo changer JSlider. When users select an instrument, the program calls instrumentBox method actionPerformed (lines 177–182) to change to the selected instrument program by invoking MidiSynthesizer method changeInstrument with the selected instrument index as the parameter.

When users drag the volume slider, the program calls volumeSlider method stateChanged (lines 201-204) to change the volume. Note that changing the volume affects only the volume of user-synthesized MIDI notes. When users drag the tempo slider, the program calls resolutionSlider’s stateChanged method (lines 225-228) to set the tempo.

Invoking method makePlaySaveButtons (lines 240–320) sets up the Play MIDI, Playback and Save buttons. Clicking Play MIDI invokes the actionPerformed method of the listenButton (lines 273–288) to playback an opened MIDI file in its entirety using class MidiData. Line 275 obtains a file from a file-chooser dialog using MidiDemo method getFile (lines 549–562). Lines 280–284 initialize and play the MIDI file using the instantiated midiData object. When the user clicks Playback, line 254 plays the recorded MIDI. This button is enabled only if some recording took place. Clicking the Save button allows the user to save the recorded sequence to a file (lines 307–310).

Method makeRecordButton (lines 323–383) creates the Record button and a listener for it. Clicking the button when it is set to recording mode (line 337) creates a new recorder using class MidiRecord (lines 339–348). If a recorder has already been created, line 351 invokes MidiRecord’s makeTrack method to make a new track for object midiRecord. When recording starts (line 353), lines 356–360 turn the recording button into a stop button and disable the playButton temporarily. When users stop the recording by clicking the recording button again, the GUI returns to its state prior to recording and the user can playback and save the MIDI sequence (lines 365–373).
“Piano Player”
The “piano player” feature of this example synchronizes the playing of a note with the highlighting of the corresponding key. The program first obtains a MIDI track from a user-specified MIDI file using class MidiData. A Timer synchronizes the MIDI events in the MIDI sequence. When the Timer reaches a MIDI event’s time stamp, it resets its delay to the time stamp of the next event to synchronize. The delay of a timer is the period of time that it waits before it causes an event.

The “piano player driver” responsible for synchronization is located in main class MidiDemo. Method makePianoPlayerButton (lines 386–436) loads the MIDI file and initializes the Timer that handles the timing of musical notes. As with the listenButton, method actionPerformed (lines 397–428) of makePlayerPianoButton uses class MidiData to load MIDI data. Lines 399–408 open a file from a file dialog box and load the MIDI data from the file. Line 410 invokes MidiData’s initializeTrack method to obtain the longest track from the loaded MIDI and to obtain the first MIDI event message from the track.

Line 414 invokes MidiData’s getResolution method (lines 67–70 in Fig. 22.7) to obtain the default tempo of the “piano player.” Lines 418–420 instantiate a new Timer object that drives the playing of notes at the time of each MIDI event. Line 419 sets the delay of the timer to be the duration of the first MidiEvent in the track by invoking MidiData’s getEventDelay method. To allow user-specified tempo changes to the “piano player,” line 424 enables the tempo slider and line 419 sets the delay of the timer to be multiplied by the value of resolution. The resolution variable, which specifies the tempo of “piano player,” is multiplied by the event interval time to obtain a new interval time used by pianoTimer. PianoTimer’s delay is set only for the first event’s duration to start the timer. Later, inner class TimerHandler (lines 439–477) method actionPerformed (lines 444–475) resets the timer’s delay to the next MidiEvent’s duration (lines 472–473). Line 426 starts the pianoTimer.

At the time of the next MIDI event, class TimerHandler’s actionPerformed method synthesizes a note from the track and “presses” a key on the piano. Using pianoTimer and its event handler, the program iterates through the track, simulating note events when it encounters appropriate MidiEvents. Inner class TimerHandler drives the synchronization by iterating through all the MidiEvents on a given track and calling method actionPerformed to play the piano. Line 451 of method actionPerformed invokes utility method noteAction (lines 481–530) to sound the note and to change the color of the specific piano key, given that the event contains a note message and that the note is within the range designator for the piano keys in this example.

Lines 484–485 of method noteAction determine whether the current MidiEvent contains a note command by invoking method getEventCommand of class MidiData. Lines 488–489 invoke MidiData method getNote to determine whether the note number specified in the noteMessage is a valid note within the range of possible piano keys, specified in constants FIRST_NOTE and MAX_KEYS (line 52).

If the command is a ShortMessage.NOTE_ON command and is within range of the piano keys, the synthesizer receives the specified note message (lines 497–498) and the corresponding piano key’s color becomes red for the duration of the event (line 494). Line 491 obtains the corresponding piano button number (lastKeyOn) that should become red.
If the command is `ShortMessage.NOTE_OFF` and is within the allowable piano key range (lines 513–517), lines 520–521 change the background of the specified piano key back to white. Lines 524-525 send the `ShortMessage` to the synthesizer so that the note will stop sounding. Because not all `NOTE_ON ShortMessage` are followed by a `NOTE_OFF ShortMessage` as one would expect, the program needs to change the last `NOTE_ON` key back to its original color at the time of the next event. For that purpose method `noteAction` assigns a `lastKeyOn` value to the last piano button invoked. The `lastKeyOn` object initialized to `-1` remains `-1` if the `NOTE_ON` command note is out of range. This limits access only to keys in range of our simulated keyboard. When `pianoTimer` reaches the next event, the program changes the background of the last “pressed” piano key back to white (lines 447–449).

When the program finishes executing method `noteAction`, line 452 invokes `MidiData` method `goNextEvent` to transition to the next event in the track. Every time the handler’s `actionPerformed` method finishes loading the next event, line 455 determines whether the next event is the last event in the track by invoking `MidiData` method `isTrackEnd`, assuming that the last event is the end-of-track `MetaEvent`. If the next event is the last event, lines 457–459 change the background color of the last “pressed” key to white and line 461 stops the `pianoTimer`. Lines 463–465 re-enable buttons that were disabled during the “piano player” feature.

22.8 Internet and World Wide Web Resources

This section presents several Internet and Web resources for the Java Media Framework and other multimedia related sites.

`java.sun.com/products/java-media/jmf/`
The *Java Media Framework home page* on the Java Web site. Here you can download the latest Sun implementation of the JMF. The site also contains the documentation for the JMF.

`www.nasa.gov/gallery/index.html`
The *NASA multimedia gallery* contains a wide variety of images, audio clips and video clips that you can download and use to test your Java multimedia programs.

`sunsite.sut.ac.jp/multimed/`
The *Sunsite Japan Multimedia Collection* also provides a wide variety of images, audio clips and video clips that you can download for educational purposes.

The *Australian National Botanic Gardens* Web site provides links to sounds of many animals. Try the *Common Birds* link.

`www.midi.com`
*Midi.com* is a MIDI resource site with a MIDI search engine, links to other MIDI sites, a list of MIDI-related books and other MIDI information.

`www.streamingmedia.com`
*Streamingmedia.com* provides many articles of the streaming media industry and technical information streaming media technology.

`www.harmony-central.com/MIDI/`
Harmony Central’s MIDI resources section contains many useful MIDI documents, links and forums that can be useful for a MIDI programmer.
22.9 (Optional Case Study) Thinking About Objects: Animation and Sound in the View

This case study has focused mainly on the elevator system model. Now that we have completed our design of the model, we turn our attention to the view, which provides the visual presentation of the model. In our case study, the view—called ElevatorView—is a JPanel object containing other JPanel “child” objects, each representing a unique object in the model (e.g. a Person, a Button, the Elevator). Class ElevatorView is the largest class in the case study. In this section, we discuss the graphics and sound classes used by class ElevatorView. We present and explain the remainder of the code in this class in Appendix I.

In Section 3.7, we constructed a class diagram for our model by locating the nouns and noun phrases from the problem statement of Section 2.7. We ignored several of these nouns, because they were not associated with the model. Now, we list the nouns and noun phrases that apply to displaying the model:

- display
- audio
- elevator music

The noun “display” corresponds to the view, or the visual presentation of the model. As described in Section 13.17, class ElevatorView aggregates several classes comprising the view. The “audio” refers to the sound effects that our simulation generates when various actions occur—we create class SoundEffects to generate these sound effects. The phrase “elevator music” refers to the music played as the Person rides in the Elevator—we create class ElevatorMusic to play this music.

The view must display all objects in the model. We create class ImagePanel to represent stationary objects in the model, such as the ElevatorShaft. We create class MovingPanel, which extends ImagePanel, to represent moving objects, such as the Elevator. Lastly, we create class AnimatedPanel, which extends MovingPanel, to represent moving objects whose corresponding images change continuously, such as a Person (we use several frames of animation to show the Person walking then pressing a button). Using these classes, we present the class diagram of the view for our simulation in Fig. 22.9.

The notes indicate the roles that the classes play in the system. According to the class diagram, class ElevatorView represents the view, classes ImagePanel, MovingPanel and AnimatedPanel relate to the graphics, and classes SoundEffects and ElevatorMusic relate to the audio. Class ElevatorView contains several instances of classes ImagePanel, MovingPanel and AnimatedPanel and one instance each of classes SoundEffects and ElevatorMusic. In Appendix I, we associate each object in the model with a corresponding class in the view.

In this section, we discuss classes ImagePanel, MovingPanel and AnimatedPanel to explain the graphics and animation. We then discuss classes SoundEffects and ElevatorMusic to explain the audio functionality.

ImagePanel
The ElevatorView uses objects from JPanel subclasses to represent and display each object in the model (such as the Elevator, a Person, the ElevatorShaft, etc.).
Class `ImagePanel` (Fig. 22.10) is a `JPanel` subclass capable of displaying an image at a given screen position. The `ElevatorView` uses `ImagePanel` objects to represent stationary objects in the model, such as the `ElevatorShaft` and the two `Floors`. Class `ImagePanel` contains an integer attribute—`ID` (line 16)—that defines a unique identifier used to track the `ImagePanel` in the view if necessary. This tracking is useful when several objects of the same class exist in the model, such as several `Person` objects. Class `ImagePanel` contains `Point2D.Double` object `position` (line 19) to represent the `ImagePanel` screen position. We will see later that `MovingPanel`, which extends `ImagePanel`, defines velocity with doubles—using type `double` yields a highly accurate velocity and position. We cast the `position` coordinates to `ints` to place the `ImagePanel` on screen (Java represents screen coordinates as `ints`) in method `setPosition` (lines 90–94). Class `ImagePanel` also contains an `ImageIcon` object called `imageIcon` (line 22)—method `paintComponent` (lines 54–60) displays `imageIcon` on screen. Lines 41–42 initialize `imageIcon` using a `String` parameter holding the name of the image. Lastly, class `ImagePanel` contains `setPanelChildren` (line 25) that stores any child objects of class `ImagePanel` (or objects of a subclass of `ImagePanel`). The child objects are displayed on top of their parent `ImagePanel`—for example, a `Person` riding inside the `Elevator`. The first method `add` (lines 63–67) appends an object to `panelChildren`. The second method `add` (lines 70–74) inserts an object into `panelChildren` at a given index. Method `setIcon` (lines 84–87) sets `imageIcon` to a new image. Objects of class `AnimatedPanel` use method `setIcon` repeatedly to change the image displayed, which causes the animation for the view—we discuss animation later in the section.

![Class diagram of elevator simulation view.](image)
package com.deitel.jhtp4.elevator.view;

public class ImagePanel extends JPanel {

    private int ID;

    private Point2D.Double position;

    private ImageIcon imageIcon;

    private Set panelChildren;

    public ImagePanel( int identifier, String imageName ) {
        super( null ); // specify null layout
        setOpaque( false ); // make transparent

        ID = identifier;

        position = new Point2D.Double( 0, 0 );
        setLocation( 0, 0 );

        imageIcon = new ImageIcon( getClass().getResource( imageName ) );

        Image image = imageIcon.getImage();
        setSize( image.getWidth( this ), image.getHeight( this ) );

        panelChildren = new HashSet();
    } // end ImagePanel constructor

Fig. 22.10 Class ImagePanel represents and displays a stationary object from the model (part 1 of 3).
// paint Panel to screen
public void paintComponent( Graphics g )
{
    super.paintComponent( g );
    // if image is ready, paint it to screen
    ImageIcon.paintIcon( this, g, 0, 0 );
}

// add ImagePanel child to ImagePanel
public void add( ImagePanel panel )
{
    panelChildren.add( panel );
    super.add( panel );
}

// add ImagePanel child to ImagePanel at given index
public void add( ImagePanel panel, int index )
{
    panelChildren.add( panel );
    super.add( panel, index );
}

// remove ImagePanel child from ImagePanel
public void remove( ImagePanel panel )
{
    panelChildren.remove( panel );
    super.remove( panel );
}

// sets current ImageIcon to be displayed
public void setIcon( ImageIcon icon )
{
    ImageIcon = icon;
}

// set on-screen position
public void setPosition( double x, double y )
{
    position.setLocation( x, y );
    setLocation( ( int ) x, ( int ) y );
}

// return ImagePanel identifier
public int getID()
{
    return ID;
}

// get position of ImagePanel
public Point2D.Double getPosition()
MovingPanel

Class MovingPanel (Fig. 22.11) is an ImagePanel subclass capable of changing its screen position according to its xVelocity and yVelocity (lines 20–21). The ElevatorView uses MovingPanel objects to represent moving objects from the model, such as the Elevator.

```java
public class MovingPanel extends ImagePanel {
    // should MovingPanel change position?
    private boolean moving;

    // number of pixels MovingPanel moves in both x and y values
    private double xVelocity;
    private double yVelocity;
}
```

Fig. 22.10 Class ImagePanel represents and displays a stationary object from the model (part 3 of 3).

Fig. 22.11 Class MovingPanel represents and displays a moving object from the model (part 1 of 3).
// constructor initializes position, velocity and image
public MovingPanel( int identifier, String imageName )
{
    super( identifier, imageName );
    // set MovingPanel velocity
    xVelocity = 0;
    yVelocity = 0;
}

// update MovingPanel position and animation frame
public void animate()
{
    // update position according to MovingPanel velocity
    if ( isMoving() ) {
        double oldXPosition = getPosition().getX();
        double oldYPosition = getPosition().getY();

        setPosition( oldXPosition + xVelocity,
                     oldYPosition + yVelocity );
    }

    // update all children of MovingPanel
    Iterator iterator = getChildren().iterator();
    while ( iterator.hasNext() ) {
        MovingPanel panel = ( MovingPanel ) iterator.next();
        panel.animate();
    }
}

// is MovingPanel moving on screen?
public boolean isMoving()
{
    return moving;
}

// set MovingPanel to move on screen
public void setMoving( boolean move )
{
    moving = move;
}

// set MovingPanel x and y velocity
public void setVelocity( double x, double y )
{
    xVelocity = x;
    yVelocity = y;
}

Fig. 22.11 Class MovingPanel represents and displays a moving object from the model (part 2 of 3).
Method `animate` (lines 35–53) moves the `MovingPanel` according to the current values of attributes `xVelocity` and `yVelocity`. If boolean variable `moving` (line 16) is `true`, lines 38–44 use attributes `xVelocity` and `yVelocity` to determine the next location for the `MovingPanel`. Lines 47–52 repeat the process for any children. In our simulation, `ElevatorView` invokes method `animate` and method `paintComponent` of class `ImagePanel` every 50 milliseconds. These rapid, successive calls move the `MovingPanel` object.

**AnimatedPanel**

Class `AnimatedPanel` (Fig. 22.12), which extends class `MovingPanel`, represents an animated object from the model (i.e., moving objects whose corresponding image changes continuously), such as a `Person`. The `ElevatorView` animates an `AnimatedPanel` object by changing the image associated with `imageIcon`.

```java
// AnimatedPanel.java
// MovingPanel subclass with animation capabilities
package com.deitel.jhtp4.elevator.view;

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

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

public class AnimatedPanel extends MovingPanel {
    // should ImageIcon cycle frames
    private boolean animating;
    // frame cycle rate (i.e., rate advancing to next frame)
    private int animationRate;
}
```

Fig. 22.11 **Class `MovingPanel`** represents and displays a moving object from the model (part 3 of 3).

Fig. 22.12 **Class `AnimatedPanel`** represents and displays an animated object from the model (part 1 of 4).
private int animationRateCounter;
private boolean cycleForward = true;

// individual ImageIcons used for animation frames
private ImageIcon imageIcons[];

// storage for all frame sequences
private java.util.List frameSequences;
private int currentAnimation;

// should loop (continue) animation at end of cycle?
private boolean loop;

// should animation display last frame at end of animation?
private boolean displayLastFrame;

// helps determine next displayed frame
private int currentFrameCounter;

// constructor takes array of filenames and screen position
public AnimatedPanel( int identifier, String imageName[] )
{
    super( identifier, imageName[ 0 ] );

    // creates ImageIcon objects from imageName string array
    imageIcons = new ImageIcon[ imageName.length ];
    for ( int i = 0; i < imageIcons.length; i++ ) {
        imageIcons[ i ] = new ImageIcon( getClass().getResource( imageName[ i ] ) );
    }

    frameSequences = new ArrayList();
}

// update icon position and animation frame
public void animate()
{
    super.animate();

    // play next animation frame if counter > animation rate
    if ( frameSequences != null && isAnimating() ) {
        if ( animationRateCounter > animationRate ) {
            animationRateCounter = 0;
            determineNextFrame();
        } else
            animationRateCounter++;
    }
} // end method animate

Fig. 22.12 Class AnimatedPanel represents and displays an animated object from the model (part 2 of 4).
// determine next animation frame
private void determineNextFrame()
{
    int frameSequence[] =
        ( int[] ) frameSequences.get( currentAnimation );

    // if no more animation frames, determine final frame,
    // unless loop is specified
    if ( currentFrameCounter >= frameSequence.length ) {
        currentFrameCounter = 0;
        
        // if loop is false, terminate animation
        if ( !isLoop() ) {
            setAnimating( false );

            if ( isDisplayLastFrame() )
            {
                // display last frame in sequence
                currentFrameCounter = frameSequence.length - 1;
            }
        }
    }
    
    // set current animation frame
    setCurrentFrame( frameSequence[ currentFrameCounter ] );
    currentFrameCounter++;
} // end method determineNextFrame

// add frame sequence (animation) to frameSequences ArrayList
public void addFrameSequence( int frameSequence[] )
{
    frameSequences.add( frameSequence );
}

// ask if AnimatedPanel is animating (cycling frames)
public boolean isAnimating()
{
    return animating;
}

// set AnimatedPanel to animate
public void setAnimating( boolean animate )
{
    animating = animate;
}

// set current ImageIcon
public void setCurrentImage( int frame )
{
    setIcon( imageIcons[ frame ] );
}
Class `AnimatedPanel` chooses the `ImageIcon` object to be drawn on screen from among several `ImageIcon` objects stored in array `imageIcons` (line 23). Class `AnimatedPanel` determines the `ImageIcon` object according to a series of `frame sequence` references, stored in `List frameSequences` (line 26). A frame sequence is an array of
integers holding the proper sequence to display the `ImageIcon` objects; specifically, each integer represents the index of an `ImageIcon` object in `imageIcons`. Figure 22.13 demonstrates the relationship between `imageIcons` and `frameSequences` (this is not a diagram of the UML). For example, frame sequence number

\[ 2 = \{ 2, 1, 0 \} \]

refers to `\{ `imageIcon[2]`, `imageIcon[1]`, `imageIcon[0]` `\}`, which yields the image sequence `\{ C, B, A `\}`. In the view, each image is a unique `.png` file. Method `addFrameSequence` (lines 102–105) adds a frame sequence to `List frameSequences`. Method `playAnimation` (lines 162–167) starts the animation associated with the parameter `frameSequence`. For example, assume an `AnimatedPanel` object called `personAnimatedPanel` in class `ElevatorView`. The code segment

```
        animatedPanel.playAnimation( 1 );
```

would generate the `\{ A, B, D, B, A `\}` image sequence using Fig. 22.13 as a reference.

Method `animate` (lines 56–70) overrides method `animate` of superclass `MovingPanel`. Lines 61–69 determine the next frame of animation depending on attribute `animationRate`, which is inversely proportional to the animation speed—a higher value for `animationRate` yields a slower frame rate. For example, if `animationRate` is 5, `animate` moves to the next frame of animation every fifth time it is invoked. Using this logic, the animation rate maximizes when `animationRate` has a value of 1, because the next frame is determined each time `animate` runs.

Method `animate` calls `determineNextFrame` (lines 73–99) to determine the next frame (image) to display—specifically, it calls method `setCurrentFrame` (lines 120–123), which sets `imageIcon` (the current image displayed) to the image returned from the current frame sequence. Lines 84–92 of `determineNextFrame` are used for “looping” purposes in the animation. If `loop` is `false`, the animation terminates after one iteration. The last frame in the sequence is displayed if `displayLastFrame is true`, and the first frame in the sequence is displayed if `displayLastFrame is false`. We explain in greater detail in Appendix I how `ElevatorView` uses `displayLastFrame` for the `Person` and `Door AnimatedPanel` s to ensure the proper display of the image. If `loop` is `true`, the animation repeats until stopped explicitly.

**Sound Effects**

We now discuss how we generate audio in our elevator simulation. We divide audio functionality between two classes—`SoundEffects` and `ElevatorMusic`. (these classes are not part of the Java packages, although `SoundEffects` uses the `java.applet` package and `ElevatorMusic` uses the `javax.sound.midi` package). Class `SoundEffects` (Figure 22.14) transforms audio (.au) and wave (.wav) files, containing such sounds as the bell ring and the person’s footsteps, into `java.applet.AudioClip` objects. In Appendix I, we list all `AudioClip`s used in our simulation. Class `ElevatorMusic` (Fig. 22.15) plays a `MIDI` (.mid) file when the person rides the elevator. The `ElevatorView` object will play the `AudioClip` and `ElevatorMusic` objects to generate sound. All sound files are in the directory structure

```
com/deitel/jhtp4/elevator/view/sounds
```
In our simulation, we use sounds and **MIDI** files provided free for download by Microsoft at the Web site:

```http
msdn.microsoft.com/downloads/default.asp
```

To download these sounds, click on “Graphics and Multimedia,” “Multimedia (General),” then “Sounds.”

---

**Fig. 22.13** Relationship between array **imageIcons** and List **frameSequences**.

(i.e., in the **sounds** directory where the classes for the view are located in the file system).

```
\[
\begin{array}{ccccccc}
0 & 1 & 2 & & & & \\
1 & 0 & 1 & 3 & 1 & 0 & \\
2 & 2 & 1 & 0 & & & \\
3 & 3 & 2 & 2 & 0 & & \\
\end{array}
\]
```

---

```java
// SoundEffects.java
// Returns AudioClip objects
package com.deitel.jhtp4.elevator.view;

// Java core packages
import java.applet.*;

public class SoundEffects {

    // location of sound files
    private String prefix = "";

    public SoundEffects() {};

    // get AudioClip associated with soundFile
    public AudioClip getAudioClip( String soundFile )
    {
        try {
            return Applet.newAudioClip( getClass().getResource( prefix + soundFile ) );
        }
        // return null if soundFile does not exist
        catch ( NullPointerException nullPointerException ) {
            return null;
        }
    }
}
```

**Fig. 22.14** Class **SoundEffects** return **AudioClip** objects (part 1 of 2).
Fig. 22.14 Class **SoundEffects** return **AudioClip** objects (part 2 of 2).

Class **SoundEffects** contains method **getAudioClip** (lines 16–27), which uses **static** method **newAudioClip** (of class **java.applet.Applet**) to return an **AudioClip** object using the **soundFile** parameter. Method **setPrefix** (lines 30–33) allows for changing the directory of a sound file (useful if we want to partition our sounds among several directories).

```
28     // set prefix for location of soundFile
29     public void setPathPrefix( String string )
30     {
31         prefix = string;
32     }
33     }
```

Fig. 22.15 Class **ElevatorMusic** plays music when a **Person** rides in the **Elevator** (part 1 of 3).

```
1     // ElevatorMusic.java
2     // Allows for MIDI playing capabilities
3     package com.deitel.jhtp4.elevator.view;
4
5     // Java core packages
6     import java.io.*;
7     import java.net.*;
8
9     // Java extension packages
10    import javax.sound.midi.*;
11
12    public class ElevatorMusic implements MetaEventListener {
13
14        // MIDI sequencer
15        private Sequencer sequencer;
16
17        // should music stop playing?
18        private boolean endOfMusic;
19
20        // sound file name
21        private String fileName;
22
23        // sequence associated with sound file
24        private Sequence soundSequence;
25
26        // constructor opens a MIDI file to play
27        public ElevatorMusic( String file )
28        {
29            // set sequencer
30            try {
31                sequencer = MidiSystem.getSequencer();
32                sequencer.addMetaEventListener( this );
33                fileName = file;
34            }
35        }
```
// handle exception if MIDI is unavailable
catch ( MidiUnavailableException midiException ) {
    midiException.printStackTrace();
}

// open music file
public boolean open() {
    try {

        // get URL for media file
        URL url = getClass().getResource( fileName );

        // get valid MIDI file
        soundSequence = MidiSystem.getSequence( url );

        // open sequencer for specified file
        sequencer.open();
        sequencer.setSequence( soundSequence );
    }

    // handle exception if URL does not exist
    catch ( NullPointerException nullPointerException ) {
        nullPointerException.printStackTrace();
        return false;
    }

    // handle exception if MIDI data is invalid
    catch ( InvalidMidiDataException midiException ) {
        midiException.printStackTrace();
        soundSequence = null;
        return false;
    }

    // handle IO exception
    catch ( java.io.IOException ioException ) {
        ioException.printStackTrace();
        soundSequence = null;
        return false;
    }

    // handle exception if MIDI is unavailable
    catch ( MidiUnavailableException midiException ) {
        midiException.printStackTrace();
        return false;
    }

    return true;
}
As we discussed in Section 22.7, Java 2 offers **MIDI** support. Class **ElevatorMusic** uses the **javax.sound.midi** package to play the **MIDI** file. Class **ElevatorMusic** listens for a **MetaMessage** event from the **MIDI** file. The sequencer generates a **MetaMessage** event. Class **ElevatorMusic**'s constructor (lines 27–40) initializes the system's **MIDI** sequencer and registers class **ElevatorMusic** for **MetaMessage** events from the sequencer. Method **open** (lines 43–85) opens the sequencer for a specified file and ensures the **MIDI** data is valid. Method **play** (lines 88–92) starts the sequencer and plays the **MIDI** file.

### Conclusion

You have completed a substantial object-oriented design (OOD) process that was intended to help prepare you for the challenges of “industrial-strength” projects. We hope you have found the optional “Thinking About Objects” sections informative and useful as a supplement to the material presented in the chapters. In addition, we hope you have enjoyed the experience designing the elevator system using the UML. The worldwide software industry has adopted the UML as the *de facto* standard for modeling object-oriented software.

Although we have completed the design process, we have merely “scratched the surface” of the implementation process. We urge you to read Appendices G, H and I on the accompanying CD, which fully implement the design. These appendices translate the UML diagrams into a 3,465-line Java program for the elevator simulation. In these appendices, we present all code that we did not cover in the “Thinking About Objects” sections and a complete “walkthrough” of this code.

1. Appendix G presents the Java files that implement events and listeners
2. Appendix H presents the Java files that implement the model
3. Appendix I presents the Java files that implement the view

We do not introduce an abundance of new material or UML design in these appendices—they simply serve to implement the UML-based diagram we have presented in previous chapters into a fully functional program. Studying the implementation in the appendices should hone the programming skills you have developed throughout the book and reinforce your understanding of the design process.

**SUMMARY**

- Through the JMF API, programmers can create Java applications that play, edit, stream and capture many popular and high-quality media types.
- JMF 2.1.1 supports popular media file types such as Microsoft Audio/Video Interleave (.avi), Macromedia Flash 2 movies (.swf), MPEG Layer 3 Audio (.mp3), Musical Instrument Digital Interface (MIDI; .mid), MPEG-1 videos (.mpeg, .mpg), QuickTime (.mov) and Sun Audio (.au).
- The Java Sound API and its extensive sound processing capabilities. Java Sound is a lower-level API that supports many of the JMF’s internal audio capabilities.
- A Player is a type of Controller in JMF that can process and play media clips. Playing media clips with interface Player can be as simple as specifying the media source, creating a Player for the media, obtaining the output media and controls GUI components from Player and displaying them. In addition, Players can access media from a capture device such as a microphone and from a Real-time transport protocol (RTP) stream—a stream of bytes sent over a network that can be buffered and played on the client computer.
- Play media involves accessing the media, creating a Controller for the media and outputting the media. Before outputting the media, there is the option of formatting the media.
- JMF provides lightweight video renderers compatible with Swing GUI components using Manager method setHint with parameter Manager.LIGHTWEIGHT_RENDERER and Boolean.TRUE.
- A MediaLocator is similar to a URL, but it also supports RTP streaming session addresses and capture device locations.
- Invoke Manager method createPlayer to create a Player object that references a media player. Method createPlayer opens the specified media source and determines the appropriate player for the media source. A NoPlayerException occurs if no appropriate player can be found for the media clip.
- Class Manager provides static methods that enable programs to access to most JMF resources.
- Throughout the media-handling process, Players generate ControllerEvents that ControllerListeners listen for. Class ControllerAdapter, which implements methods of interface ControllerListener.
- Controllers use state transitions to confirm their position in the media processing algorithm.
- Player's realize method to confirm all resources necessary to play media. Method realize places the Player in a Realizing state where the player interacts with its media sources. When a Player completes realizing, it generates a RealizedCompleteEvent—a type of ControllerEvent that occurs when a Player completes its transition to state Realized.
- Player method prefetch causes the Player to obtain hardware resources for playing the media and begin buffering the media data. Buffering the media data reduces the delay before the media clip plays because media reading can take a long time.
• Invoke `Player` method `getVisualComponent` method to obtain the visual component of a video. Invoke `Player` method `getControlPanelComponent` to return the player’s GUI controls.

• When the media clip ends, the `Player` generates a `ControllerEvent` of type `EndOfMediaEvent`.

• `Player` method `setMediaTime` sets the position of the media to a specific time in the media.

• Invoking `Player` method `start` starts media playback. It also buffers and realizes the player if that has not been done.

• Capture devices such as microphones have the ability to convert analog media into digitized media. This type of media is known as captured media.

• Class `DataSource` abstracts the media source to allow a program to manipulate it and provides a connection to the media source.

• Interface `Processor` allows a program to manipulate data at the various processing stages. It extends interface `Player` and provides more control over media processing.

• Monitoring allows you to hear or see the captured media as it is captured and saved. A `MonitorControl` and other control objects obtained from `Controller` by invoking method `getControl`.

• JMF provides class `Format` to describe the attributes of a media format, such as the sampling rate (which controls the quality of the sound) and whether the media should be in stereo or mono format. `FormatControl` objects that allow us to format objects that support format controls.

• Class `CaptureDeviceManager` enables a program to access capture device information.

• A `CaptureDeviceInfo` object provides the essential information necessary of a capture device’s `DataSource`.

• Invoke `Manager` method `createDataSource` to obtain the `DataSource` object that of that media location.

• `Manager` method `createRealizedProcessor` creates a realized `Processor` object that can start processing media data. The method requires as an argument a `ProcessorModel` object containing the specifications of the `Processor`.

• Use a `ContentDescriptor` to describe the content-type of output from a `Processor`. `FileTypeDescriptor` specifies the a file media content.

• Call `Processor` method `getTrackControls` to get each track’s controls.

• An object that implements interface `DataSink` enables media data to be output to a specific location—most commonly a file. `Manager` method `createDataSink` receives the `DataSource` and `MediaLocator` as arguments to create a `DataSink` object.

• Register a `DataSinkListener` to listen for `DataSinkEvents` generated by a `DataSink`. A program can call `DataSinkListener` method `dataSinkUpdate` when each `DataSinkEvent` occurs. A `DataSink` causes an `EndOfStreamEvent` when the capture stream connection closes.

• Streaming media refers to media that is transferred from a server to a client in a continuous stream of bytes. Streaming media technology loads media data into buffers before displaying media.

• JMF provides a streaming media package that enables Java applications to send and receive streams of media in the formats discussed earlier in this chapter. JMF uses the industry-standard Real-Time Transport Protocol (RTP) to control media transmission. RTP is designed specifically to transmit real-time media data.

• Use a `DataSink` or a `RTPManager` to stream media. `RTPManager`s provide more control and versatility for the transmission. If an application sends multiple streams, the application must have
an **RTPManager** for each separate streaming session. Both require the **DataSource** obtained from **Processor**’s `getOutput` method.

- The URL of RTP streams is in format: `rtp://<host>:<port>/<contentType>`

- Formatting the media can only be done when the **Processor** has been configured. To notify the program when it completes **Processor** configuration, register a **ControllerListener** to notify the program that it has completed configuring. A **ConfigureCompleteEvent** occurs when **Processor** completes configuration.

- **Processor** method setContentTypeDescriptor sets the stream to an RTP-enabled format with **ContentDescriptor.RAW_RTP** parameter.

- **TrackControl** interface allow the formats of the media tracks to be set.

- **SessionAddress** contains an IP addresses and port number used in the streaming process. **RTPManager**s use **SessionAddress**es to stream media.

- Invoke **RTPManager** method `initialize` to initialize the local streaming session with the local session address as the parameter. Invoke **RTPManager** method `addTarget` to add the destination session address as the client recipient of the media stream. To stream media to multiple clients, call **RTPManager** method `addTarget` for each destination address.

- **RTPManager** method `removeTargets` closes streaming to specific destinations. **RTPManager** method `dispose` release the resources held by the RTP sessions.

- The Java Sound API provides classes and interfaces for accessing, manipulating and playing Musical Instrumental Data Interface (MIDI) and sampled audio.

- A sound card is required to play audio with Java Sound. Java Sound throws exceptions when it accesses audio system resources to process audio on a computer that does not have a sound card.

- Programmers can use package `javax.sound.sampled` to play sampled audio file formats, which includes Sun Audio (.au), Wave (.wav) and AIFF (.aiff).

- To process audio data, we can use a **Clip** line that allows the flow of raw digital data to audio data we can listen to.

- An **AudioInputStream** object to point to the audio stream. Class **AudioInputStream** (a subclass of **InputStream**) provides access to the audio stream contents.

- The length of video and audio clips is measured in frames. Each frame represents data at a specific time interval in the audio file.

- The algorithm for playing sampled audio supported by Java Sound is as follows: obtain an **AudioInputStream** from an audio file, obtain a formatted **Clip** line, load the **AudioInputStream** into the **Clip** line, start the data flow in the **Clip** line.

- All **Lines** generate **LineEvents** that can be handled by **LineListener**. The first step to sampled audio playback involves obtaining the audio stream from an audio file.

- Class **AudioSystem** enables a program to access many audio system resources required to play and manipulate sound files.

- Method `getAudioInputStream` throws an **UnsupportedAudioFileException** if the specified sound file is a non-audio file or contains a sound clip format that is not supported by Java Sound.

- Method `getLine` requires a **Line.Info** object as an argument, which specifies the attributes of the line the **AudioSystem** should obtain.

- We can use a **DataLine.Info** object that specifies a **Clip** data line, a general encoding format and a buffer range. We need to specify a buffer range so the program can determine the best buffer size given a preferred range.
• DataLine.Info objects specify information about a Clip line, such as the formats supported by the Clip. The DataLine.Info object constructor receives as arguments the Line class, the line’s supported AudioFormat, the minimum buffer size and the maximum buffer size in bytes.

• AudioSystem method getLine and Clip method open throw LineUnavailableExceptions if another application is using the requested audio resource. Clip method open also throws an IOException if open is unable to read the specified AudioInputStream.

• Invoke Clip method start to begin audio playback.

• When a LineEvent occurs, the program calls LineListener method update to process the event. The four LineEvent types are defined in class LineEvent.Type. The event types are CLOSE, OPEN, START and STOP.

• Method close of class Line stops audio activity and closes the line—which releases any audio resources obtained previously by the Line.

• Clip method loop can be called with parameter Clip.LoopCONTINUOUSLY to replay the audio clip forever.

• Invoking method stop of interface Clip only stops data activity in the Line. Invoking method start resumes data activity.

• MIDI (Musical Instrument Digital Interface) music can be created through a digital instrument, such as an electronic keyboard (synthesizer), or through packaged software synthesizers. A MIDI synthesizer is a device that can produce MIDI sounds and music.

• The MIDI specification provides detailed information on the formats of a MIDI file. For detailed information on MIDI and its specification, visit their official Web site at www.midi.org. Java Sound’s MIDI package allows developers to access the data that specify the MIDI, but it does not provide support for the specification.

• Interpretation of MIDI data varies between synthesizers and will sound different with different instruments. Package javax.sound.midi enables program to manipulate, play and synthesize MIDI. There are three MIDI types—0 (the most common), 1 and 2. Java Sound supports MIDI files with .mid extensions and .rmf (Rich Music Format).

• Some file parsers in various operating systems are unable to interpret the MIDI file as a MIDI file that Java can play.

• MIDI playback is accomplished by a MIDI sequencer. Specifically, sequencers can play and manipulate a MIDI sequence, which is the data formula that tells a device how to handle the MIDI data.

• Often, MIDI is referred to as a sequence, because the musical data in MIDI is composed of a sequence of events. The simplicity of MIDI data enables us to view each event individually and learn the purpose of each event. The process to MIDI playback involves accessing a sequencer, loading a MIDI sequence or a MIDI file into a sequencer and starting the sequencer.

• Method getSequence also can obtain a MIDI sequence from a URL or an InputStream. Method getSequence throws an InvalidMidiDataException if the MIDI system detects an incompatible MIDI file.

• Interface Sequencer, which extends interface MidiDevice (the super-interface for all MIDI devices), represents the standard device to play MIDI data.

• Sequencer’s open method prepares to play a Sequence. Sequencer method setSequence loads a MIDI Sequence into the Sequencer and throws an InvalidMidiException if the Sequencer detects an unrecognizable MIDI sequence. Sequencer method play begins playing the MIDI sequence.
• A MIDI track is a recorded sequence of data; MIDIs usually contain multiple tracks. MIDI tracks are similar to CD tracks except that the music data in MIDI are played simultaneously. Class Track (package javax.sound.midi) provides access to the MIDI music data stored in the MIDI tracks.

• MIDI data in MIDI tracks are represented by MIDI events. MIDI events are the holders of the MIDI action and the time when the MIDI command should occur. There are three types of MIDI message—ShortMessage, SysexMessage and MetaMessage. ShortMessages provide instructions, such as specific notes play, and can configure options, such when a MIDI starts. The other two less-used messages are exclusive system messages called SysexMessage and MetaMessages which may tell a device that the MIDI has reached the end of a track. This section deals exclusively with ShortMessages that play specific notes. Each MidiMessage is encapsulated in a MidiEvent and a sequence of MidiEvents form a MIDI track.

• Each MidiEvent’s getTick method provides the time when the event takes place (time stamp).

• ShortMessage method getCommand returns the command integer of the message. ShortMessage method getData1 returns the first status byte of the message. ShortMessage method getData2 returns the second status byte. The first and second status bytes vary in interpretation according to the type of command in ShortMessage.

• General MIDI recording is accomplished through a sequencer. Interface Sequencer provides simple methods for recording—assuming the transmitters and receivers of MIDI devices are “wired” correctly.

• After setting up a sequencer and an empty sequence, a Sequencer object can invoke its startRecording method to enable and start recording on the empty track. Method recordEnable of interface Sequencer takes a Track object and a channel number as the parameters to enable recording on a track.

• Method write of class MidiSystem writes the sequence to a specified file.

• An alternative method to record MIDI without having to deal with transmitters and receivers is to create events from ShortMessages. The events should be added to a track of a sequence.

• Interface Synthesizer is a MidiDevice interface which enables access to MIDI sound generation, instruments, channel resources, and sound banks.

• A SoundBank is the container for various Instruments, which tell the computer how to sound a specific note and are programmed algorithms of instructions. Different notes on various instruments are played through a MidiChannel on different tracks simultaneously to produce symphonic melodies.

• Acquiring any MIDI resources throws a MidiUnavailableException if the resource is unavailable.

• Invokes Synthesizer’s getChannels method to obtain all 16 channels from the synthesizer. A MidiChannel can sound a note by calling its noteOn method with the note number (0-127) and volume as parameters. MidiChannel’s noteOff method turns off a note with just the note number parameter.

• Synthesizer’s getAvailableInstruments method obtains the default instrument programs of a synthesizer. One can also import more instruments by loading a customized sound bank through method loadAllInstruments (SoundBank) in interface Synthesizer. A sound bank usually has 128 instruments. MidiChannel’s programChange method loads the desired instrument program into the synthesizer.

• Invoke Receiver’s send method with a MidiMessage and a time stamp as its parameters to send MIDI message to all its transmitters.
TERMINOLOGY

addControllerListener method of Controller
addDataSinkListener method of DataSink
addLineListener method of Line
addMetaEventListener method of Sequencer
addTarget method of RTPManager
AudioFormat class
AudioFormat.Encoding.PCM_SIGNED
AudioInputStream class
AudioSystem class
Boolean.TRUE
broadband
CannotRealizeException class
capture
capture device
captured media
CaptureDevice interface
CaptureDeviceInfo class
CaptureDeviceManger class
Clip interface
Clip.class
Clip.Loop_CONTINUOUSLY
Clock interface
close method of Controller
close method of Line
close method of MidiDevice
configure method of Processor
configureComplete method of ControllerAdapter
ConfigureCompleteEvent class
Controller.Prefetching
Controller.Prefetched
Controller.Realized
Controller.Realizing
Controller.Started
Controller.Stopped
ControllerAdapter class
ControllerEvent class
ControllerListener interface
createDataSink method of Manager
createDataSource method of Manager
createPlayer method of Manager
createProcessor method of Manager
createRealizedProcessor method of Manager
createSendStream method of RTPManager
createTrack method of Sequence

DataLine interface
DataLine.Info class
DataSink interface
DataSinkEvent class
DataSinkListener interface
dataSinkUpdate method of
DataSinkListener
DataSource class
deleteTrack method of Sequence
device ports
Direct Sound
dispose method of RTPManager
encoding
endOfMedia method of
ControllerAdapter
EndOfMediaEvent class
EndofStreamEvent class
FileTypeDescriptor class
FileTypeDescriptor.QUICKTIME
Format class
FormatControl interface
frames
get method of Track
getAudioInputStream method of
AudioSystem
getBank method of Patch
getChannels method of Synthesizer
getCommand method of ShortMessage
getControlComponent method of Control
getControlPanelComponent method of Player
getData1 method of ShortMessage
ggetData2 method of ShortMessage
getDataOutput method of Processor
getDeviceList method of
CaptureDeviceManager
getFormat method of AudioFormat
getFormat method of FormatControl
getFormatControls method of
CaptureDevice
getFrameLength method of
AudioInputStream
getFrameSize method of AudioFormat
getLine method of AudioSystem
getLocator method of
CaptureDeviceInfo
getMessage method of MidiEvent
getProgram method of Patch
getReceiver method of MidiDevice
getResolution method of Sequence
getSequence method of MidiSystem
getSequencer method of MidiSystem
getSupportedFormats method of FormatControl
getSynthesizer method of MidiSystem
getTargetFormat method of AudioSystem
getTick method of MidiEvent
getTrackControls method of Processor
getTracks method of Sequence
getTransmitter method of MidiDevice
getType method of LineEvent
getVisualComponent method of Player
initialize method of RTPManager
Instrument class
InvalidMidiException class
InvalidSessionAddress class
isEnabled method of FormatControl
isLineSupported method of AudioSystem

Java Media Framework
Java Sound
javax.media package
javax.media.control package
javax.media.datasink package
javax.media.format package
javax.media.protocol package
javax.media.rtp package
javax.sound.midi package
javax.sound.sampled package
JOptionPane package
JOptionPane
JOptionPane
JOptionPane
Line interface
LineEvent class
LineEvent.Type.STOP
LineListener interface
LineUnavailableException class
loop method of Clip
Manager class
Manager.LIGHTWEIGHT_RENDERER
media
media clip
media location
media tracks
MediaLocator class
MIDI (Musical Instrument Digital Interface)
MIDI Specification
MidiChannel interface
MidiEvent class
MidiMessage class
MidiSystem class
MidiUnavailableException class
Mixers
MonitorControl interface
monitoring
MP3
MPEG-1
network ports
newInstance method of RTPManager
NoDataSourceException class
NoDataSinkException class
NoPlayerException class
NoProcessorException class
noteOff method of MidiChannel
noteOn method of MidiChannel
open method of Clip
open method of DataSink
open method of MidiDevice
output format
packetized data
Patch class
pitch
Player interface
pre-buffer
prefetchComplete method of ControllerAdapter
PrefetchCompleteEvent class
pre-process
Processor interface
Processor.Configured
Processor.Configuring
ProcessorModel class
propagation delay
protocol
QuickTime
realize method of Controller
realizeComplete method of ControllerAdapter
RealizeCompleteEvent class
Receiver interface
recordEnable method of Sequencer
removeTargets method of RTPManager
RMF (Rich Music Format)
RTP (Real-time Transport Protocol)
RTPManager class
SecurityException class
send method of Receiver
SendStream interface
SELF-REVIEW EXERCISES

22.1 Fill in the blanks in each of the following.

a) Class ________ provides access to many JMF resources.
b) In addition to locations of media files stored on the local computer, a ________ can also specify the location of capture devices and RTP sessions.
c) Class ________ provides access to sampled audio system resources while class ________ provides access to MIDI system resources.
d) An event of type ________ indicates that a Controller has establish communications with the media source.
e) Method createRealizedProcessor takes a ________ as an argument.
f) In order, the Processor’s states are: Unrealized, __________, __________, __________, __________, __________ and Started.
g) Constant ________ specifies that the Processor should output media in QuickTime format.
h) To stream media, we can use a ________ or a ________.
i) ________ objects set the stream formats for capture devices.
j) Invoking Clip method ________ with constant ________ as an argument replays a sampled audio file continuously.
k) A MIDI ________ contains multiple tracks, which contain a sequence of MIDI ________ that each encapsulate a MIDI ________.

22.2 State whether each of the following is true or false. If false, explain why.

a) Manager method setHint can be used to specify that the visual component of a media clip should be rendered using lightweight GUI components.
b) A ControllerListener handles events generated by a DataSink.
c) Only objects that implement interface Processor can play media.
d) A **Player** cannot access media from capture devices; a **Processor** must be used for this purpose.

e) A **Clip** plays MIDI **Sequences**.

f) MIDI playback stops automatically when the **Sequencer** reaches the end of a MIDI **Sequence**.

g) An **RTPManager** can stream an entire media file regardless of the number of tracks in the file.

h) Method **createPlayer** throws a **NoDataSourceException** if it is unable to locate the specified media data source.

**ANSWERS TO SELF-REVIEW EXERCISES**

22.1  

a) **Manager**.  
b) **MediaLocator**.  
c) **AudioSystem**, **MidiSystem**.  
d) **RealizeCompleteEvent**.  
e) **ProcessorModel**.  
f) **Configuring**, **Configured**, **Realizing**, **Realized**, **Prefetching**, **Prefetched**.  
g) **FileTypeDescriptor**.**QUICKTIME**.  
h) **DataSink**, **RTPManager**.  
i) **FormatControl**.  
j) **loop**, **Clip.LOOP_CONTINUOUSLY**.  
k) **Sequence**, events, message.

22.2  
The answers to Self-Review Exercise 3.2 are as follows:

a) True.

b) False. A **DataSinkListener** handles **DataSinkEvent**s generated by a **DataSink**.

c) False. Objects that implement **Player** or **Processor** can play media.

d) False. Both a **Processor** and a **Player** can access media from capture devices.

e) False. A **Sequencer** plays MIDI sequences.

f) True.

g) False. Each **RTPManager** can stream only one track.

h) False. Method **createPlayer** throws a **NoPlayerException** if it is unable to locate the specified media data source.

**EXERCISES**

22.3  
Wave audio clips are commonly used to play sounds that alert the user of a problem in a program. Typically, such sounds are accompanied by error-message dialogs. Modify the **DivideByZeroTest** example of Fig. 14.1 to play an error-message sound (in addition to displaying an error message dialog) if the user enters an invalid integer or attempts to divide by zero. Preload a compatible sound clip using a **Clip** line as demonstrated in Fig. 22.5. The **Clip** line needs to support only the format of the chosen sound clip. There should be a separate method that invokes the playback of the clip. When the program detects an exception, it should call this method to play the error message sound. After each clip playback, the program needs to rewind the clip by invoking **Clip**’s method **setFramePosition** with the frame position as the argument, so that the clip can replay from its beginning position.

22.4  
Incorporate MIDI file playback capabilities, as demonstrated in class **MidiData** (Fig. 22.7), into the **ClipPlayer** demo. Class **ClipPlayer** should have separate methods for obtaining MIDI sequence data and for playing back the sequence with a sequencer.

22.5  
The **SimplePlayer** demo (Fig. 22.7) demonstrated JMF’s media playback (videos, capture media) capabilities using interface **Player**. Using the **SimplePlayer** demo as a guideline, develop a karaoke application in which one portion of the program plays a music/video file (preferably without lyrics) while another portion of the program simultaneously captures the user’s voice. The program should start playback and capture as soon as it obtains the media. It is important that the program allows the user to control both the capture and music, so the control GUls of each media should be displayed. When the media file finishes playing, the voice capture should cease and the program
should reset the music to the beginning. The program should close all Player-related resources when the user terminates the program.

22.6 Modify your solution to Exercise 3.5 by implementing the program using interface Processor. Create a Processor that is ready to display media with no format or output specifications. The voice capture should not end and the media should not rewind when the media finishes playing. Deallocate Processor-related resources when the user opens a new file or closes the program. All other program details remain the same as specified in Exercise 3.5.

22.7 Referring to the file saving process demonstrated in class CapturePlayer (Fig. 22.2), modify your solution to Exercise 22.6 by saving both audio streams to two separate QuickTime files. Specify the media tracks to be in AudioFormat.IMA4 encoding. The program should display file saver dialogs for each audio file saver and a message when the saving completes or stops. There should be separate data writers for each audio stream. The program should close the data writers when there is no more data to process or when the user terminates the program. DataSinkEvent method getSourceDataSink is available to obtain the DataSink that is generating the DataSinkEvent. Use MonitorControls to monitor both audio streams, so there is no need to display video or default user controls. MonitorControl method setEnabled is available to enable monitoring of the audio streams. Display one of the MonitorControls in a dialog box. Make sure to close Processor resources when there is no more data from the media file or when the user opens another file or terminates the program.

22.8 Modify class MidiSynthesizer (Fig. 22.9) into an application where the user can play violin notes by pressing the computer keyboard keys. Use the virtual key code of the keys to specify the note number that the synthesizer should play. The violin’s program number is 40. Use the first and ninth channel to sound the notes at the same time. The ninth channel may generate a different version of the violin notes.

SPECIAL SECTION: CHALLENGING MULTIMEDIA PROJECTS

The preceding exercises are keyed to the text and designed to test the reader’s understanding of fundamental JMF and Java Sound concepts. This section includes a collection of advanced multimedia projects. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects. [Note: Solutions are not provided for these exercises.]

22.9 Modify the ClipPlayer (Fig. 22.5, Fig. 22.6) demo to provide a replay checkbox that allows the user to replay the sampled audio file.

22.10 Modify the RTPServer demo (Fig. 22.3) to enable transmission of the audio portions of media files to two clients. The application testing class should have twice the number of IP and port number inquiry dialog boxes. The program can check for audio formats by matching track control formats of the media to be instances of AudioFormat (class AudioFormat).

22.11 Many Web sites are able to play video clips. The SimplePlayer (Fig. 22.1) program can be an applet. Simplify the SimplePlayer program to an applet that plays a preloaded media clip on a web page. Insert this applet tag into your HTML file:

```html
<applet code = AppletName.class width = # height = # >
<param name = file value = "sample.mov" />
</applet>
```

22.12 Modify class MidiRecord (Fig. 22.8) and class MidiData (Fig. 22.7) to create a new application class that duplicates MIDI files by recording the sequence to a new file. Play the sequence
using MidiData and use its Transmitter to transmit MIDI information to the Receiver of MidiRecord.

22.13 Modify your solution to Exercise 22.6 to create a streaming karaoke application in which the application streams only the video portion of a music video and the sound stream is replaced by a voice capture stream. [Note: If the video format contains only one track for both audio and video, the application cannot choose to stream only the video portion of the track.]

22.14 Modify class MidiData (Fig. 22.7) to load all tracks of a MIDI file and revise class MidiDemo (Fig. 22.10) to enable the user to select the playback of each individual track displayed in a JList selector panel. Allow the user to replay the sequence forever.

22.15 Implement an MP3 player with a file list window using Vectors and a JList.

22.16 Modify class MidiRecord (Fig. 22.8) and class MidiDemo (Fig. 22.10) to allow the user to record MIDI to individual tracks stored in a Vector. Playback of the recorded MIDI should play all MIDI tracks simultaneously.

22.17 Currently the MidiDemo program (Section 22.7) records synthesized music with the first available instrument (i.e., Grand Piano). Modify class MidiDemo so that music will be recorded with a user-selected instrument, and allow the user to change the instrument during recording. Also allow the user to import their own sound banks. Make changes to classes MidiSynthesizer, MidiData and MidiRecord as needed. (Hint: The command parameter to change instrument is ShortMessage.PROGRAM_CHANGE)

22.18 Modify the SimplePlayer demo (Fig. 22.1) to support multiple media players. Present each media clip in its own JInternalFrame. The program needs to create separate Players for each media clip and should register a ControllerListener for each player. ControllerEvent method getSourceController is available to obtain the controller generating the ControllerEvent. Implement the program using a dynamic data structure such as a Vector to store the multiple Players.

22.19 Modify your solution to Exercise 3.7 to save both media streams to one file and play the combined stream. Use Manager method createMergingDataSource, which receives an array of DataSource objects, to save both the capture stream and the music stream into one stream, whose content-type will be MIXED. The program should obtain the output DataSource objects from the Processors as the DataSource objects to be merged. The program must also obtain a duplicate DataSource (of the merged DataSource) for creating the Player for that DataSource. To do this, use Manager method createCloneableDataSource to create a Cloneable DataSource with the merged DataSource as the argument. Duplicate the DataSource for the player by invoking method createClone of interface Cloneable on the DataSource (similar to obtaining the FormatControls of a CaptureDevice DataSource in the CapturePlayer demo (Fig. 22.2)).

22.20 A program can record MIDI without the use of transmitters and receivers by manually creating MidiMessages, placing them in MidiEvents, and adding these events to a track. In addition to the MidiMessage argument, a program must specify a time stamp to create the MidiEvent, expressed in ticks (i.e. milliseconds of long type), so the program must obtain the system's current time in milliseconds, which can be obtained from System method currentTimeMillis. Track method add is available to add events to a track. Create an acoustics table (e.g. drums, cymbals, etc.) where the user can selected a sequence of instruments to play. Allow the user to save the recorded MIDI sequence to a file.

22.21 Create a peer-to-peer teleconferencing kit that enables users to talk to and hear each other. To listen to each other, each user must open an RTP session for the capture stream. A program can open an RTP stream with a MediaLocator specifying the RTP session address. Then, the program
can use the `MediaLocator` to create a `Player` for the RTP stream. The program can send the voice capture as demonstrated in the `RTPManager` demo (Fig. 22.3). To send the voice capture to more than two people, use `RTPManagers` for each separate session and call its `addTarget` method to add each recipient’s session address as a destination of the capture stream. This is referred to as `multicast-unicast` sessions.

22.22 Using the “piano player” driver (Section 22.7.4 discusses the driver) from class `MidiDemo` (Fig. 22.10) and image animation functions from Chapter 18, write a program that displays a bouncing ball whose pinnacle height is specified by the note numbers (of a `ShortMessage` in a `MidiEvent`) of a loaded track in a MIDI file. The program should use the duration ticks of the `MidiEvent` as the duration of the bounce.

22.23 Most karaoke music videos are in MPEG format, for which JMF provides an `MpegAudioControl` interface to control the audio channels. For multilingual and karaoke MPEG videos, channel layers direct the main audio to one of two audio streams (e.g. english-dubbed audio stream) or both audio streams (e.g. both music and song channels turned on in karaoke videos). As an additional feature for MPEG videos to your solution to Exercise 3.6, obtain and display the control GUI component from a `MpegAudioControl`, if any, or display a customized radio button GUI selector to let the user select the channel layout of the MPEG audio. Interface `MpegAudioControl` provides method `setChannelLayout` to set the audio channel layout for MPEG videos, with the channel layout as the argument. (Refer to the use of `MonitorControl` in Fig. 22.2)

22.24 Create a multimedia-rich Tic-Tac-Toe game that plays sampled audio sounds when a player makes a valid move or makes an invalid move. Use a `Vector` to store preloaded `AudioInputStreams` representing each audio clip. Use a `Clip` line to play sounds in response to the user interactions with the game. Play continuous MIDI background music while the game is in progress. Use the JMF’s interface `Player` to play a video when a player wins.

22.25 Java Sound’s MIDI package can access MIDI software devices as well as hardware devices. If you have a MIDI keyboard that the computer can detect or one that can be plugged into a MIDI IN port of a sound card, Java Sound can access that keyboard. Use synthesizers, receivers, transmitters and sequencers to allow the user to record MIDI synthesized by the electronic keyboard. `MidiSystem` method `getMidiDeviceInfo` is available to obtain all detectable MIDI devices’ information (array of `MidiDevice.Info` objects). Use `MidiSystem` method `getMidiDevice` with a `MidiDevice.Info` argument to obtain the specified MIDI device resource.

22.26 Enhance the `MidiDemo` program to allow for more configurations such as tempo, control and replay. `Sequencers` can implement `MetaEventListener`s to handle `MetaMessages` and `ControlEventListener`s to handle `ShortMessage.CONTROL_CHANGE` commands (see the MIDI specification for the type of changes and the various `MetaEvents`). Interface `Sequencer` also provides many configuration and control methods that affect sequencer playback.

22.27 Enhance the `RTPServer` program (Fig. 22.3, Fig. 22.4) to a video distribution server where a live video feed from a video capture device (e.g., VFW TV card, digital cameras) is broadcast to everyone in the network. Use the IP address ending with .255 (i.e. to create a `SessionAddress`) to broadcast to everyone in the network subnet. A server program should have access to stream transmission status, error controls and delay management. Package `javax.media.rtp` provides many interfaces to handle stream configurations and statistics. Package `javax.media.rtp` allows the user to access RTP session reports. Package `javax.media.rtp.event` contains many events generated during an RTP session that can be used to perform RTP enhancements at those stages in the session. Package `javax.media.control` provides several control interfaces useful in RTP sessions.

22.28 Enhance the media player application solution to Exercise 3.18 by adding editing features to the program. First add a replay checkbox. Although a `Processor` is more suitable for control tasks than a `Player` is, one can also use both interfaces by creating a `Player` for the output `Data-
Source from the Processor. Control features should include track formatting, frame control setting (interface FrameRateControl), buffer control (interface BufferControl), and quality control (interface QualityControl). Include a program option that allows the user to save media clips given the settings of these controls. For capture devices, there is a PortControl interface available to control their ports of devices. These interfaces are in package javax.media.control. In package javax.media, there are other interfaces such as Codec and Effect that enable further rendering and processing of media to specific media formats. Allow the user to import new codecs. Also implement an editing feature which enables the user to extract certain portions of a media clip. (Hint: set the media position of a media clip and obtain the Processor output at the marked positions.)

22.29 Package javax.media.sound offers many audio system resources. Use this package to create a sound capture program that allows the users to save the capture stream in their desired formats, bit rates, frequencies and encodings.

22.30 Create a visualization studio that displays graphics bars that synchronize with sampled audio playback.

22.31 Extend MP3 playback support for class ClipPlayer (Fig. 22.5) using classes in package javax.media.sound.spi. The MP3 encoding process uses the Huffman algorithm.

22.32 (Story Teller) Record audio for a large number of nouns, verbs, articles, prepositions, etc. Then use random number generation to forms sentences and have your program speak the sentences.

22.33 (Project: Multimedia Authoring System) Develop a general-purpose multimedia authoring system. Your program should allow the user to form multimedia presentations consisting of text, audios, images, animations and eventually, videos. Your program lets the user weave together a presentation consisting of any of these multimedia elements that are selected from a catalog your program displays. Provide controls to allow the user to customize the presentation dynamically as the presentation is delivered.

22.34 (Video Games) Video games have become wildly popular. Develop your own Java video game program. Have a contest with your classmates to develop the best original video game.

22.35 (Physics Demo: Bouncing Ball) Develop an animated program that shows a bouncing ball. Give the ball a constant horizontal velocity. Allow the user to specify the coefficient of restitution, e.g., a coefficient of restitution of 75% means that after the ball bounces it returns to only 75% of its height before it was bounced. Your demo should take gravity into effect—this will cause the bouncing ball to trace a parabolic path. Track down a “boing” sound (like a spring bouncing) and play the sound every time the ball hits the ground.

22.36 (Physics Demo: Kinetics) If you have taken physics, implement a Java program that will demo concepts like energy, inertia, momentum, velocity, acceleration, friction, coefficient of restitution, gravity and others. Create visual effects and use audios for emphasis and realism.

22.37 (Project: Flight Simulator) Develop your own flight simulator Java program. This is a very challenging project. It is also an excellent candidate for a contest with your classmates.

22.38 (Towers of Hanoi) Write an animated version of the Towers of Hanoi problem we presented in Exercise 6.37. As each disk is lifted off a peg or slid onto a peg play a “whooshing” sound. As each disk lands on the pile play a “clunking” sound. Play some appropriate background music.

22.39 (Tortoise and the Hare) Develop a multimedia version of the Tortoise and Hare simulation we presented in Exercise 7.41. You might record an announcer’s voice calling the race, “The contenders are at the starting line,” “And they’re off!” “The Hare pulls out in front.” “The Tortoise is coming on strong,” etc. As the race proceeds, play the appropriate recorded audios. Play sounds to simulate the animals’ running, and don’t forget the crowd cheering! Do an animation of the animals racing up the side of the slippery mountain.
22.40  (Knight’s Tour Walker) Develop multimedia-based versions of the Knight’s Tour programs you wrote in Exercises 7.22 and 7.23.

22.41  (Pinball Machine) Here’s another contest problem. Develop a Java program that simulates a pinball machine of your own design. Have a contest with your classmates to develop the best original multimedia pinball machine. Use every possible multimedia trick you can think of to add “pizzazz” to your pinball game. Try to keep the game mechanisms close to those of real pinball games.

22.42  (Roulette) Study the rules for the game of roulette and implement a multimedia-based version of the game. Create an animated spinning roulette wheel. Use audio to simulate the sound of the ball jumping the various compartments that correspond to each of the numbers. Use an audio to simulate the sound of the ball falling into its final slot. While the roulette wheel is spinning, allow multiple players to place their bets. When the ball lands in its final slot, you should update the bank accounts of each of the players with the appropriate wins or losses.

22.43  (Craps) Simulate the complete game of craps. Use a graphical representation of a craps table. Allow multiple players to place their bets. Use an animation of the player who is rolling the dice and show the animated dice rolling eventually to a stop. Use audio to simulate some of the chatter around the craps table. After each roll, the system should update the bank accounts of each of the players depending on the bets they have made.

22.44  (Morse Code) Modify your solution to Exercise 10.26 to output the morse code using audio clips. Use two different audio clips for the dot and dash characters in Morse code.
Java Demos

A.1 Introduction

In this appendix, we list some of the best Java demos we found on the Web. We began our journey at software.dev.earthweb.com/java. This site is an incredible Java resource and has some of the best Java demos including a huge compilation of games. The code ranges from basic to complex. Many of the authors of these games and other resources have provided source code. We hope you enjoy surfing these sites as much as we did.

A.2 The Sites

software.dev.earthweb.com/java

Gamelan, a site owned by EarthWeb, has been a wonderful Java resource since the early days of Java. This site originally was a large Java repository where individuals traded ideas on Java and examples of Java programming. One of its early benefits was the volume of Java source code that was available to the many people learning Java. It is now an all-around Java resource with Java references, free Java downloads, areas where you can ask questions to Java experts, discussion groups on Java, a glossary of Java-related terminology, upcoming Java-related events, directories for specialized industry topics and hundreds of other Java resources.

www.jars.com

Another EarthWeb Web site is JARS—originally called the Java Applet Rating Service. The JARS site calls itself the “#1 Java Review Service.” This site originally was a large Java repository for applets. It rated every applet registered at the site as top 1%, top 5% and top 25%, so you could immediately view the best applets on the Web. Early in the development of the Java language, having your applet rated here was a great way to demonstrate your Java programming abilities. JARS is now another all-around resource for Java programmers.

1. There are many Java-related Web sites that cover more advanced Java topics such as servlets, Java-Server Pages (JSP), Enterprise Java Beans (EJB), database, Java 2 Enterprise Edition (J2EE), Java 2 Micro Edition (J2ME), Security, XML and many more. These sites are provided in our companion book, Advanced Java 2 Platform How to Program (ISBN# 0-13-089560-1).
www.javashareware.com
Java Shareware contains hundreds of Java applications, applets, classes and other Java resources. The site includes a large number of links to other Java developer Web sites.

javaboutique.internet.com
Java Boutique is a great resource for any Java programmer. The site offers applets, applications, forums, tutorials, reviews, glossaries and more.

www.thejmaker.com
The J Maker has numerous live Java demos, including games, menus and animated visual effects. Many of the programs on this site are acclaimed worldwide and some have won awards.

www.demicron.com/gallery/photoalbum2/index.shtml
The PhotoAlbum II has some of the coolest effects we have found on the Web. The effects include a liquid effect, folding an image into a paper airplane and others.

www.blaupunkt.de/simulations/svdef_en.html
The Sevilla RDM 168 is a simulation, built with Java, of a car Radio and CD player. You can tune into a variety of on-line radio stations or CDs, adjust the volume, clock settings and more.

www.frontiernet.net/~imaging/play_a_piano.html
Play A Piano is a Java applet that allows you to play the piano or watch the sound waves and listen as the piano plays itself.

www.gamesdomain.co.uk/GamesArena/goldmine
Goldmine is a fun game using simple animation.

www.javaonthebrain.com
This site offers many new and original Java games for the Web. Source code for many of the programs is provided.

www.javagamepark.com
If you are looking for games, the Java Game Park site has loads of them. All of the games are written in Java. The source code is provided in some cases.

teamball.sdsu.edu/~boyns/java/
This site has interesting games written in Java and includes source code for many of them.

www.cruzio.com/~sabweb/arcade/index.html
SABames Arcade is another source for Java video games. Source code is provided for many of the games. Don’t miss the SabBowl bowling game.

dogfeathers.com/java/hyprcube.html
Stereoscopic Animated Hypercube. If you happen to possess the old red and blue 3D glasses, check this site out. The programmer was able to create a 3D image using Java. It isn’t really a complicated image, just some cubes, however the idea that you can create images that will jump off your screen is a great concept!

www.npac.syr.edu/projects/vishuman/VisibleHuman.html
This site has won various awards. You can look at cross sections of the human body.

www-groups.dcs.st-and.ac.uk/~history/Java/
Famous Curves Applet Index. Provides graphs of complex curves. Allows the user to alter the parameters to the equations that calculate the curves.
Java Resources

B.1 Resources

java.sun.com
The Sun Microsystems, Inc. Java Web site is an essential stop when searching the Web for Java information. Go to this site to download the Java 2 Software Development Kit. This site offers news, information, on-line support, code samples and more.

java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
This site reviews the Sun Microsystems, Inc. code conventions for the Java programming language.

www.softwaredev.earthweb.com
This site provides a wide variety of information on Java and other Internet-related topics. The Java directory page contains links to thousands of Java applets and other Java resources.

softwaredev.earthweb.com/java
The Gamelan site has been a wonderful Java resource since the early days of Java. This site originally was a large Java repository where individuals traded ideas on Java and examples of Java programming. One of its early benefits was the volume of Java source code that was available to the many people learning Java. It is now an all-around Java resource with Java references, free Java downloads, areas where you can ask questions of Java experts, discussion groups on Java, a glossary of Java terminology, upcoming Java events, directories for specialized industry topics and hundreds of other Java resources.

www.jars.com
Another Earthweb.com Web site is JARS—originally called the Java Applet Rating Service. The JARS site calls itself the “#1 Java Review Service.” This site originated as a large Java repository for applets. Its benefit was that it rated every applet registered at the site as top 1%, top 5% and top 25%, so you could immediately view the best applets on the Web. Early in the development of the Java language, having your applet rated here was a great way to demonstrate your Java programming abilities. JARS is now another all-around resource for Java programmers.

1. There are many Java-related Web sites that cover more advanced Java topics such as servlets, Java-Server Pages (JSP), Enterprise Java Beans (EJB), database, Java 2 Enterprise Edition (J2EE), Java 2 Micro Edition (J2ME), Security, XML and many more. These sites are provided in our companion book, Advanced Java 2 Platform How to Program (ISBN# 0-13-089560-1).
developer.java.sun.com/developer
On the Sun Microsystems Java Web site, visit the Java Developer Connection. The site includes technical support, discussion forums, on-line training courses, technical articles, resources, announcements of new Java features, early access to new Java technologies and links to other important Java Web sites. Even though the site is free, you must register to use it.

www.acme.com/java
This page has several animated Java applets with the source code provided. This site is an excellent resource for information on Java. The page provides software, notes and a list of hyperlinks to other resources. Under software you will find animated applets, utility classes and applications.

www.javaworld.com/index.html
The JavaWorld online magazine provides a collection of Java articles, tips, news and discussions. A questions area of this site addresses both general and specific problems programmers face.

www.nikos.com/javatoys
The Java Toys Web site includes links to the latest Java news, Java User Groups (JUGs), FAQs, tools, Java-related mailing lists, books and white papers.

www.java-zone.com
The Development Exchange Java Zone site includes Java discussion groups, an “ask the Java Pro” section and some recent Java news.

www.ibiblio.org/javafaq
This site provides the latest Java news. It also has some helpful Java resources including a Java FAQ List, a tutorial called Brewing Java, Java User Groups, Java Links, the Java Book List, Java Trade Shows, Java Training and Exercises.

dir.yahoo.com/Computers_and_Internet/Programming_Languages/Java
Yahoo, a popular World Wide Web search engine, provides a complete Java resource. You can initiate a search using key words or explore the categories listed at the site including games, contests, events, tutorials and documentation, mailing lists, security and more.

www-106.ibm.com/developerworks/java
The IBM Developers Java Technology Zone site lists the latest news, tools, code, case studies and events related to IBM and Java.

### B.2 Products

java.sun.com/products
Download the Java 2 SDK and other Java products from the Sun Microsystems Java Products page.

www.sun.com/forte/ffj/index.cgi
The NetBeans IDE is a customizable, platform independent, visual programming development environment.

www.borland.com/jbuilder
The Borland JBuilder IDE home page has news, product information and customer support.

www.towerj.com
At this site you will find information on how to enhance the performance of server-side Java applications along with free evaluation copies of native Java compilers.

www.symantec.com/domain/cafe
Visit the Symantec site for information on their Visual Café Integrated Development Environment.

www-4.ibm.com/software/ad/vajava
Download and read more about the IBM Visual Age for Java development environment.
www.metrowerks.com
The Metrowerks CodeWarrior IDE supports a few programming languages, including Java.

B.3 FAQs

www.ibiblio.org/javafaq/javafaq.html
This is a comprehensive resource for both Java language basics and more advanced topics in Java Programming. Section 6: Language Issues and Section 11: Common Errors and Problems may be particularly useful, as they clarify situations that often are not explained well.

www.afu.com/javafaq.html
This is another FAQ which covers a fairly broad slice of topics in Java. This includes some good code samples and hints for getting projects to compile and run.

www.nikos.com/javatoys/faqs.html
The Java Toys Web site includes links to FAQs on a broad range of Java-related topics.

www.jguru.com/faq/index.jsp
This is a thorough compilation of FAQs on Java and related subjects. Questions can be read in order or searched for by subject.

B.4 Tutorials

Several tutorials are on the sites listed in the Resources section.

java.sun.com/docs/books/tutorial/
The Java Tutorial Site has a number of tutorials, including sections on JavaBeans, JDBC, RMI, Servlets, Collections and Java Native Interface.

www.ibiblio.org/javafaq/
This site provides the latest Java news. It also has some helpful Java resources including the Java FAQ List, a tutorial called Brewing Java, Java User Groups, Java Links, the Java Book List, Java Conferences, Java Course Notes and Java Seminar Slides.

B.5 Magazines

www.javaworld.com
The JavaWorld on-line magazine is an excellent resource for current Java information. You will find news clips, conference information and links to Java-related Web sites.

www.sys-con.com/java
Catch up with the latest Java news at the Java Developer’s Journal site. This magazine is one of the premier resources for Java news.

www.javareport.com
The Java Report is a great resource for Java developers. You will find the latest industry news, sample code, event listings, products and jobs.

www.javapro.com
The JAVAPro is an excellent Java developer resource with up-to-the-minute technical articles.

B.6 Java Applets

java.sun.com/applets/index.html
This page contains a variety of Java applet resources, including free applets you can use on your own Web site, the demonstration applets from the J2SDK and a variety of other applets. There is a section entitled “Applets at Work” where you can read about applets in industry.
developer.java.sun.com/developer/
On the Sun Microsystems Java Web site, visit the Java Developer Connection. This free site has close to one million members. The site includes technical support, discussion forums, on-line training courses, technical articles, resources, announcements of new Java features, early access to new Java technologies, and links to other important Java Web sites.

www.gamelan.com
Gamelan, a site owned by EarthWeb, has been a wonderful Java resource since the early days of Java. This site originally was a large Java repository where individuals traded ideas on Java and examples of Java programming. One of its early benefits was the volume of Java source code that was available to the many people learning Java. It is now an all-around Java resource with Java references, free Java downloads, areas where you can ask questions to Java experts, discussion groups on Java, a glossary of Java-related terminology, upcoming Java-related events, directories for specialized industry topics and hundreds of other Java resources.

www.jars.com
Another EarthWeb Web site is JARS—originally called the Java Applet Rating Service. The JARS site calls itself the “#1 Java Review Service.” This site originally was a large Java repository for applets. Its benefit was that it rated every applet registered at the site as top 1%, top 5% and top 25%, so you could immediately view the best applets on the Web. Early in the development of the Java language, having your applet rated here was a great way to demonstrate your Java programming abilities. JARS is now another all-around resource for Java programmers.

B.7 Multimedia

java.sun.com/products/java-media/jmf
This is the Java Media Framework home page on the Java Web site. Here you can download the latest Sun implementation of the JMF (see Chapter 22). The site also contains the documentation for the JMF.

www.nasa.gov/gallery/index.html
The NASA multimedia gallery contains a wide variety of images, audio clips and video clips that you can download and use to test your Java multimedia programs.

sunsite.sut.ac.jp/multimed
The Sunsite Japan Multimedia Collection also provides a wide variety of images, audio clips and video clips that you can download for educational purposes.

The Australian National Botanic Gardens Web site provides links to the audio clips of many animal sounds.

java.sun.com/products/java-media/jmf/index.html
This site provides an HTML-based on-line guide to the Java Media Framework API.

B.8 Newsgroups

Newsgroups are forums on the Internet in which people can post questions, responses, hints and clarifications for other users. Newsgroups can be a valuable resource to anyone learning Java or anyone with questions on specific Java topics. When posting your own questions to a newsgroup, provide specific details of the problem you are trying to solve (such as a problem with a program you are writing). This will enable other people reading the newsgroup to understand your posting and (hopefully) provide a response to you. Be sure to specify a subject heading that clearly states your problem. If you are responding to other people’s
questions, verify your response before posting it to ensure that the response is correct. Newsgroups should not be used to promote products or services, nor should contact information (such as email addresses for other newsgroup users) be used for unrelated purposes. These are generally not forums for chatting, so posts should be courteous and to the point.

Normally, newsgroup reader software is required for you to interact with a newsgroup. Such software is provided as part of Netscape Navigator and Microsoft Outlook Express, and there are many other newsgroup software programs available. You also can access newsgroups through the Web. If you do not use newsgroups already, try the Google site

`groups.google.com`

Type the name of the group you would like to view (you can also search for newsgroups) and you will be presented with a list of that newsgroup’s current topics and questions. Listed below are some Java newsgroups that you may find useful.

`news:comp.lang.java.advocacy`
This group is an active center of discussion for current Java culture, including the merits of different programming languages.

`news:comp.lang.java.announce`
This group provides announcements on major additions to Java, new class libraries and conferences.

`news:comp.lang.java.api`
This group contains questions about bugs, compile errors, Java specifications and which classes are most appropriate for different situations.

`news:comp.lang.java.gui`
This group responds to problems encountered working with Java graphical user interfaces. If you are having trouble with a particular component, layout or event, this may be a good place to start.

`news:comp.lang.java.help`
This group is particularly active. It addresses many language and environment issues. Questions include requests for classes or algorithms that solve a specific problem.

`news:comp.lang.java.machine`
For people interested in the inner workings of Java, this group focuses on the Java Virtual Machine.

`news:comp.lang.java.misc`
This group contains everything from job postings to questions about the Java documentation and is mainly for questions that do not fit into other Java newsgroup categories.

`news:comp.lang.java.programmer`
This group is another extremely active forum that addresses a range of questions. Posts tend to be largely project-oriented and concerned with overall program style and structure.

`news:comp.lang.java.softwaretools`
This newsgroup is centered around Java software products, their uses, their faults and possible modifications. Some questions about writing effective software are also included here.

`news:comp.lang.java.tech`
This is a new group devoted to the technical aspects of Java and its inner workings.
Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>parentheses</td>
<td>left to right</td>
</tr>
<tr>
<td>[]</td>
<td>array subscript</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>member selection</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>unary postincrement</td>
<td>right to left</td>
</tr>
<tr>
<td>--</td>
<td>unary postdecrement</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>unary preincrement</td>
<td>right to left</td>
</tr>
<tr>
<td>--</td>
<td>unary predecrement</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>unary plus</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>unary logical negation</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>unary bitwise complement</td>
<td></td>
</tr>
<tr>
<td>( type )</td>
<td>unary cast</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise left shift</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise right shift with sign extension</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>bitwise right shift with zero extension</td>
<td></td>
</tr>
</tbody>
</table>

Fig. C.1  Operator precedence chart (part 1 of 2).
<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>relational less than</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relational less than or equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;</td>
<td>relational greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>relational greater than or equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>instanceof</td>
<td>type comparison</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>relational is equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>!=</td>
<td>relational is not equal to</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive OR</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>boolean logical exclusive OR</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise inclusive OR</td>
</tr>
<tr>
<td></td>
<td>boolean logical inclusive OR</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>ternary conditional</td>
<td>right to left</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>right to left</td>
</tr>
<tr>
<td>+=</td>
<td>addition assignment</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>subtraction assignment</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>multiplication assignment</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>division assignment</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>modulus assignment</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>bitwise AND assignment</td>
<td></td>
</tr>
<tr>
<td>^=</td>
<td>bitwise exclusive OR assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>bitwise inclusive OR assignment</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>bitwise left shift assignment</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>bitwise right shift with sign extension assignment</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;</td>
<td>bitwise right shift with zero extension assignment</td>
<td></td>
</tr>
</tbody>
</table>

Fig. C.1 Operator precedence chart (part 2 of 2).
**ASCII Character Set**

The digits at the left of the table are the left digits of the decimal equivalent (0-127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for “F” is 70, and the character code for “&” is 38.

Most users of this book are interested in the ASCII character set used to represent English characters on many computers. The ASCII character set is a subset of the Unicode character set used by Java to represent characters from most of the world’s languages. For more information on the Unicode character set, see Appendix K.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>nul</td>
<td>soh</td>
<td>stx</td>
<td>etx</td>
<td>eot</td>
<td>enq</td>
<td>ack</td>
<td>bel</td>
<td>bs</td>
<td>ht</td>
</tr>
<tr>
<td>nl</td>
<td>vt</td>
<td>ff</td>
<td>cr</td>
<td>so</td>
<td>si</td>
<td>dle</td>
<td>dc1</td>
<td>dc2</td>
<td>dc3</td>
</tr>
<tr>
<td>dc4</td>
<td>nak</td>
<td>syn</td>
<td>etb</td>
<td>can</td>
<td>em</td>
<td>sub</td>
<td>esc</td>
<td>fs</td>
<td>gs</td>
</tr>
<tr>
<td>rs</td>
<td>us</td>
<td>sp</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>‘</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
</tr>
<tr>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
</tr>
<tr>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>del</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Objectives

- To understand basic number systems concepts such as base, positional value, and symbol value.
- To understand how to work with numbers represented in the binary, octal, and hexadecimal number systems.
- To be able to abbreviate binary numbers as octal numbers or hexadecimal numbers.
- To be able to convert octal numbers and hexadecimal numbers to binary numbers.
- To be able to covert back and forth between decimal numbers and their binary, octal, and hexadecimal equivalents.
- To understand binary arithmetic, and how negative binary numbers are represented using two’s complement notation.

Here are only numbers ratified.
William Shakespeare

Nature has some sort of arithmetic-geometrical coordinate system, because nature has all kinds of models. What we experience of nature is in models, and all of nature’s models are so beautiful.

It struck me that nature’s system must be a real beauty, because in chemistry we find that the associations are always in beautiful whole numbers—there are no fractions.
Richard Buckminster Fuller
E.1 Introduction

In this appendix, we introduce the key number systems that Java programmers use, especially when they are working on software projects that require close interaction with “machine-level” hardware. Projects like this include operating systems, computer networking software, compilers, database systems, and applications requiring high performance.

When we write an integer such as 227 or -63 in a Java program, the number is assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10. Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.

As we will see, binary numbers tend to be much longer than their decimal equivalents. Programmers who work in assembly languages and in high-level languages like Java that enable programmers to reach down to the “machine level,” find it cumbersome to work with binary numbers. So two other number systems the octal number system (base 8) and the hexadecimal number system (base 16)—are popular primarily because they make it convenient to abbreviate binary numbers.

In the octal number system, the digits range from 0 to 7. Because both the binary number system and the octal number system have fewer digits than the decimal number system, their digits are the same as the corresponding digits in decimal.

The hexadecimal number system poses a problem because it requires sixteen digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15. Thus in hexadecimal we can have numbers like 876 consisting solely of decimal-like digits, numbers like 8A55F consisting of digits and letters, and numbers like FFE consisting solely of letters. Occasionally, a hexadecimal number spells a common word such as FACE or FEED—this can appear strange to programmers accustomed to working with numbers.

Each of these number systems uses positional notation—each position in which a digit is written has a different positional value. For example, in the decimal number 937 (the 9, the 3, and the 7 are referred to as symbol values), we say that the 7 is written in the ones position, the 3 is written in the tens position, and the 9 is written in the hundreds position.
Notice that each of these positions is a power of the base (base 10), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.3).

<table>
<thead>
<tr>
<th>Binary digit</th>
<th>Octal digit</th>
<th>Decimal digit</th>
<th>Hexadecimal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. E.1** Digits of the binary, octal, decimal and hexadecimal number systems.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Lowest digit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Highest digit</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>F</td>
</tr>
</tbody>
</table>

**Fig. E.2** Comparing the binary, octal, decimal and hexadecimal number systems.

**Positional values in the decimal number system**

<table>
<thead>
<tr>
<th>Decimal digit</th>
<th>Position name</th>
<th>Positional value</th>
<th>Positional value as a power of the base (10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Hundreds</td>
<td>100</td>
<td>$10^2$</td>
</tr>
<tr>
<td>3</td>
<td>Tens</td>
<td>10</td>
<td>$10^1$</td>
</tr>
<tr>
<td>7</td>
<td>Ones</td>
<td>1</td>
<td>$10^0$</td>
</tr>
</tbody>
</table>

**Fig. E.3** Positional values in the decimal number system.
For longer decimal numbers, the next positions to the left would be the thousands position (10 to the 3rd power), the ten-thousands position (10 to the 4th power), the hundred-thousands position (10 to the 5th power), the millions position (10 to the 6th power), the ten-millions position (10 to the 7th power), and so on.

In the binary number 101, we say that the rightmost 1 is written in the ones position, the 0 is written in the twos position, and the leftmost 1 is written in the fours position. Notice that each of these positions is a power of the base (base 2), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.4).

For longer binary numbers, the next positions to the left would be the eights position (2 to the 3rd power), the sixteens position (2 to the 4th power), the thirty-twos position (2 to the 5th power), the sixty-fours position (2 to the 6th power), and so on.

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position, and the 4 is written in the sixty-fours position. Notice that each of these positions is a power of the base (base 8), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.5).

For longer octal numbers, the next positions to the left would be the five-hundred-and-twelves position (8 to the 3rd power), the four-thousand-and-ninety-sixes position (8 to the 4th power), the thirty-two-thousand-seven-hundred-and-sixty eights position (8 to the 5th power), and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position, and the 3 is written in the two-hundred-and-fifty-sixes position. Notice that each of these positions is a power of the base (base 16), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.6).

<table>
<thead>
<tr>
<th>Positional values in the binary number system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary digit</td>
</tr>
<tr>
<td>Position name</td>
</tr>
<tr>
<td>Positional value</td>
</tr>
<tr>
<td>Positional value as a power of the base (2)</td>
</tr>
</tbody>
</table>

Fig. E.4 Positional values in the binary number system.

<table>
<thead>
<tr>
<th>Positional values in the octal number system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal digit</td>
</tr>
<tr>
<td>Position name</td>
</tr>
<tr>
<td>Positional value</td>
</tr>
<tr>
<td>Positional value as a power of the base (8)</td>
</tr>
</tbody>
</table>

Fig. E.5 Positional values in the octal number system.
For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16 to the 3rd power), the sixty-five-thousand-five-hundred-and-thirty-six position (16 to the 4th power), and so on.

E.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers

The main use for octal and hexadecimal numbers in computing is for abbreviating lengthy binary representations. Figure E.7 highlights the fact that lengthy binary numbers can be expressed concisely in number systems with higher bases than the binary number system.

<table>
<thead>
<tr>
<th>Decimal number</th>
<th>Binary representation</th>
<th>Octal representation</th>
<th>Hexadecimal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>
A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2). Consider the following 12-digit binary number and its octal and hexadecimal equivalents. See if you can determine how this relationship makes it convenient to abbreviate binary numbers in octal or hexadecimal. The answer follows the numbers.

<table>
<thead>
<tr>
<th>Binary Number</th>
<th>Octal equivalent</th>
<th>Hexadecimal equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011010001</td>
<td>4321</td>
<td>8D1</td>
</tr>
</tbody>
</table>

To see how the binary number converts easily to octal, simply break the 12-digit binary number into groups of three consecutive bits each, and write those groups over the corresponding digits of the octal number as follows

100 011 010 001

4 3 2 1

Notice that the octal digit you have written under each group of thee bits corresponds precisely to the octal equivalent of that 3-digit binary number as shown in Fig. E.7.

The same kind of relationship may be observed in converting numbers from binary to hexadecimal. In particular, break the 12-digit binary number into groups of four consecutive bits each and write those groups over the corresponding digits of the hexadecimal number as follows

1000 1101 0001

8 D 1

Notice that the hexadecimal digit you wrote under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number as shown in Fig. E.7.

### E.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting these groups as their equivalent octal digit values or hexadecimal digit values. This process may be used in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101, and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101, and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

### E.4 Converting from Binary, Octal, or Hexadecimal to Decimal

Because we are accustomed to working in decimal, it is often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is “really” worth. Our diagrams in Section E.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its
positional value, and sum these products. For example, the binary number 110101 is converted to decimal 53 as shown in Fig. E.8.

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values as shown in Fig. E.9.

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values as shown in Fig. E.10.

### E.5 Converting from Decimal to Binary, Octal, or Hexadecimal

The conversions of the previous section follow naturally from the positional notation conventions. Converting from decimal to binary, octal, or hexadecimal also follows these conventions.

Suppose we wish to convert decimal 57 to binary. We begin by writing the positional values of the columns right to left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

<table>
<thead>
<tr>
<th>Converting a binary number to decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Positional values:</strong></td>
</tr>
<tr>
<td><strong>Symbol values:</strong></td>
</tr>
<tr>
<td><strong>Products:</strong></td>
</tr>
<tr>
<td><strong>Sum:</strong></td>
</tr>
</tbody>
</table>

Fig. E.8 Converting a binary number to decimal.

<table>
<thead>
<tr>
<th>Converting an octal number to decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Positional values:</strong></td>
</tr>
<tr>
<td><strong>Symbol values:</strong></td>
</tr>
<tr>
<td><strong>Products</strong></td>
</tr>
<tr>
<td><strong>Sum:</strong></td>
</tr>
</tbody>
</table>

Fig. E.9 Converting an octal number to decimal.

<table>
<thead>
<tr>
<th>Converting a hexadecimal number to decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Positional values:</strong></td>
</tr>
<tr>
<td><strong>Symbol values:</strong></td>
</tr>
<tr>
<td><strong>Products</strong></td>
</tr>
<tr>
<td><strong>Sum:</strong></td>
</tr>
</tbody>
</table>

Fig. E.10 Converting a hexadecimal number to decimal.
Appendix E

Number Systems (on CD) 1363

Positional values:  64  32  16  8  4  2  1

Then we discard the column with positional value 64 leaving:

Positional values:  32  16  8  4  2  1

Next we work from the leftmost column to the right. We divide 32 into 57 and observe that there is one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. We divide 16 into 25 and observe that there is one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We divide 8 into 9 and observe that there is one 8 in 9 with a remainder of 1. The next two columns each produce quotients of zero when their positional values are divided into 1 so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1 so we write 1 in the 1 column. This yields:

Positional values:  32  16  8  4  2  1
Symbol values:  1  1  1  0  0  1

and thus decimal 57 is equivalent to binary 111001.

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:  512  64  8  1

Then we discard the column with positional value 512, yielding:

Positional values:  64  8  1

Next we work from the leftmost column to the right. We divide 64 into 103 and observe that there is one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe that there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder so we write 7 in the 1 column. This yields:

Positional values:  64  8  1
Symbol values:  1  4  7

and thus decimal 103 is equivalent to octal 147.

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:  4096  256  16  1

Then we discard the column with positional value 4096, yielding:

Positional values:  256  16  1

Next we work from the leftmost column to the right. We divide 256 into 375 and observe that there is one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe that there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder so we write 7 in the 1 column. This yields:
Positional values: 256 16 1
Symbol values: 1 7 7

and thus decimal 375 is equivalent to hexadecimal 177.

E.6 Negative Binary Numbers: Two’s Complement Notation

The discussion in this appendix has been focussed on positive numbers. In this section, we explain how computers represent negative numbers using *two’s complement notation*. First we explain how the two’s complement of a binary number is formed, and then we show why it represents the negative value of the given binary number.

Consider a machine with 32-bit integers. Suppose

```java
int value = 13;
```

The 32-bit representation of `value` is

```
00000000 00000000 00000000 00001101
```

To form the negative of `value` we first form its *one’s complement* by applying Java’s bitwise complement operator (~):

```java
onesComplementOfValue = ~value;
```

Internally, `~value` is now `value` with each of its bits reversed—ones become zeros and zeros become ones as follows:

```
value: 00000000 00000000 00000000 00001101
~value (i.e., value’s one’s complement): 11111111 11111111 11111111 11110010
```

To form the two’s complement of `value` we simply add one to `value`’s one’s complement. Thus

```
Two’s complement of value:
11111111 11111111 11111111 11110011
```

Now if this is in fact equal to -13, we should be able to add it to binary 13 and obtain a result of 0. Let us try this:

```
00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-------------------------------
00000000 00000000 00000000 00000000
```

The carry bit coming out of the leftmost column is discarded and we indeed get zero as a result. If we add the one’s complement of a number to the number, the result would be all 1s. The key to getting a result of all zeros is that the twos complement is 1 more than the one’s complement. The addition of 1 causes each column to add to 0 with a carry of 1. The carry keeps moving leftward until it is discarded from the leftmost bit, and hence the resulting number is all zeros.

Computers actually perform a subtraction such as

```java
x = a - value;
```
by adding the two’s complement of value to a as follows:

\[ x = a + (-\text{value} + 1); \]

Suppose \( a \) is 27 and \( \text{value} \) is 13 as before. If the two’s complement of value is actually the negative of \( \text{value} \), then adding the two’s complement of value to \( a \) should produce the result 14. Let us try this:

<p>| |
|  |<br />
|---|---|---|---|---|---|---|---|
| a (i.e., 27) | 00000000 | 00000000 | 00000000 | 00011011 |</p>
<table>
<thead>
<tr>
<th>+(-value + 1)</th>
<th>+11111111</th>
<th>11111111</th>
<th>11111111</th>
<th>11110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000110</td>
<td></td>
</tr>
</tbody>
</table>

which is indeed equal to 14.

**SUMMARY**

- When we write an integer such as 19 or 227 or -63 in a Java program, the number is automatically assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10.

- Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.

- The octal number system (base 8) and the hexadecimal number system (base 16) are popular primarily because they make it convenient to abbreviate binary numbers.

- The digits of the octal number system range from 0 to 7.

- The hexadecimal number system poses a problem because it requires sixteen digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15.

- Each number system uses positional notation—each position in which a digit is written has a different positional value.

- A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2).

- To convert an octal number to a binary number, simply replace each octal digit with its three-digit binary equivalent.

- To convert a hexadecimal number to a binary number, simply replace each hexadecimal digit with its four-digit binary equivalent.

- Because we are accustomed to working in decimal, it is convenient to convert a binary, octal or hexadecimal number to decimal to get a sense of the number’s “real” worth.

- To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value, and sum these products.

- Computers represent negative numbers using two’s complement notation.

- To form the negative of a value in binary, first form its one’s complement by applying Java’s bitwise complement operator (~). This reverses the bits of the value. To form the two’s complement of a value, simply add one to the value’s one’s complement.
TERMINOLOGY

<table>
<thead>
<tr>
<th>base</th>
<th>digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>base 2 number system</td>
<td>hexadecimal number system</td>
</tr>
<tr>
<td>base 8 number system</td>
<td>negative value</td>
</tr>
<tr>
<td>base 10 number system</td>
<td>octal number system</td>
</tr>
<tr>
<td>base 16 number system</td>
<td>one’s complement notation</td>
</tr>
<tr>
<td>binary number system</td>
<td>positional notation</td>
</tr>
<tr>
<td>bitwise complement operator (∼)</td>
<td>positional value</td>
</tr>
<tr>
<td>conversions</td>
<td>symbol value</td>
</tr>
<tr>
<td>decimal number system</td>
<td>two’s complement notation</td>
</tr>
</tbody>
</table>

SELF-REVIEW EXERCISES

E.1 The bases of the decimal, binary, octal, and hexadecimal number systems are ________, ________, ________, and ________ respectively.

E.2 In general, the decimal, octal, and hexadecimal representations of a given binary number contain (more/fewer) digits than the binary number contains.

E.3 (True/False) A popular reason for using the decimal number system is that it forms a convenient notation for abbreviating binary numbers simply by substituting one decimal digit per group of four binary bits.

E.4 The (octal / hexadecimal / decimal) representation of a large binary value is the most concise (of the given alternatives).

E.5 (True/False) The highest digit in any base is one more than the base.

E.6 (True/False) The lowest digit in any base is one less than the base.

E.7 The positional value of the rightmost digit of any number in either binary, octal, decimal, or hexadecimal is always ________.

E.8 The positional value of the digit to the left of the rightmost digit of any number in binary, octal, decimal, or hexadecimal is always equal to ________.

E.9 Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

<table>
<thead>
<tr>
<th>decimal</th>
<th>hexadecimal</th>
<th>binary</th>
<th>octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>...</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>100</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>...</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

E.10 Convert binary 110101011000 to octal and to hexadecimal.

E.11 Convert decimal 177 to binary, to octal, and to hexadecimal.

E.12 Convert binary 1101110 to decimal.

E.13 Convert octal 317 to decimal.

E.14 Convert hexadecimal FACE to binary.

E.15 Convert octal 7316 to binary.

E.16 Convert hexadecimal 4FEC to decimal. (Hint: First convert 4FEC to binary then convert that binary number to octal.)

E.17 Convert hexadecimal EFD4 to decimal.

E.18 Show the binary representation of decimal 417. Then show the one’s complement of 417, and the two’s complement of 417.
What is the result when the one's complement of a number is added to itself?

**SELF-REVIEW ANSWERS**

E.1 10, 2, 8, 16.

E.2 Fewer.

E.3 False.

E.4 Hexadecimal.

E.5 False. The highest digit in any base is one less than the base.

E.6 False. The lowest digit in any base is zero.

E.7 1 (the base raised to the zero power).

E.8 The base of the number system.

E.9 Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

<table>
<thead>
<tr>
<th>decimal</th>
<th>1000</th>
<th>100</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>hexadecimal</td>
<td>4096</td>
<td>256</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>binary</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>octal</td>
<td>512</td>
<td>64</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

E.10 Octal 6530; Hexadecimal D58.

E.11 Binary 1111 1101 1100 1110.

E.12 Binary 111 011 111 110.

E.13 Binary 0 100 111 111 101 100; Octal 47754.

E.14 Decimal 2+4+8+32+64=110.

E.15 Decimal 7+1*8+3*64=7+8+192=207.

E.16 Decimal 4+13*16+15*256+14*4096=61396.

E.17 Decimal 177 to binary:

\[
\begin{align*}
256 & \quad 128 & \quad 64 & \quad 32 & \quad 16 & \quad 8 & \quad 4 & \quad 2 & \quad 1 \\
128 & \quad 64 & \quad 32 & \quad 16 & \quad 8 & \quad 4 & \quad 2 & \quad 1 \\
\end{align*}
\]

\[
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1) = 10110001
\]

E.17 to octal:

\[
\begin{align*}
512 & \quad 64 & \quad 8 & \quad 1 \\
64 & \quad 8 & \quad 1 \\
(2*64)+(6*8)+(1*1) = 261
\end{align*}
\]

E.17 to hexadecimal:

\[
\begin{align*}
256 & \quad 16 & \quad 1 \\
16 & \quad 1 \\
(11*16)+(1*1) = B1
\end{align*}
\]
E.18 Binary:

\[
\begin{align*}
512 & \quad 256 & \quad 128 & \quad 64 & \quad 32 & \quad 16 & \quad 8 & \quad 4 & \quad 2 & \quad 1 \\
256 & \quad 128 & \quad 64 & \quad 32 & \quad 16 & \quad 8 & \quad 4 & \quad 2 & \quad 1 \\
\end{align*}
\]

\[(1 \times 256) + (1 \times 128) + (0 \times 64) + (1 \times 32) + (0 \times 16) + (0 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1)\]

110100001

One’s complement: 001011110
Two’s complement: 001011111

Check: Original binary number + its two’s complement

\[
\begin{align*}
110100001 \\
001011111 \\
\hline
000000000
\end{align*}
\]

E.19 Zero.

**EXERCISES**

E.20 Some people argue that many of our calculations would be easier in the base 12 number system because 12 is divisible by so many more numbers than 10 (for base 10). What is the lowest digit in base 12? What might the highest symbol for the digit in base 12 be? What are the positional values of the rightmost four positions of any number in the base 12 number system?

E.21 How is the highest symbol value in the number systems we discussed related to the positional value of the first digit to the left of the rightmost digit of any number in these number systems?

E.22 Complete the following chart of positional values for the rightmost four positions in each of the indicated number systems:

<table>
<thead>
<tr>
<th>decimal</th>
<th>10000100 10 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>base 6</td>
<td>.... 6...</td>
</tr>
<tr>
<td>base 13</td>
<td>... 169...</td>
</tr>
<tr>
<td>base 3</td>
<td>27...........</td>
</tr>
</tbody>
</table>

E.23 Convert binary 100101111010 to octal and to hexadecimal.

E.24 Convert hexadecimal 3A7D to binary.

E.25 Convert hexadecimal 765F to octal. (Hint: First convert 765F to binary, then convert that binary number to octal.)

E.26 Convert binary 1011110 to decimal.

E.27 Convert octal 426 to decimal.

E.28 Convert hexadecimal FFFF to decimal.

E.29 Convert decimal 299 to binary, to octal, and to hexadecimal.

E.30 Show the binary representation of decimal 779. Then show the one’s complement of 779, and the two’s complement of 779.

E.31 What is the result when the two’s complement of a number is added to itself?

E.32 Show the two’s complement of integer value -1 on a machine with 32-bit integers.
Objectives

• To introduce the javadoc J2SDK tool.
• To introduce documentation comments.
• To understand javadoc tags.
• To be able to generate HTML API documentation with javadoc.
• To understand javadoc generated documentation files.

Oh I get by with a little help from my friends.
John Lennon and Paul McCartney

I feel
The link of nature draw me.
John Milton

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

There is only one religion, though there are a hundred versions of it.
George Bernard Shaw

What I like in a good author is not what he says, but what he whispers.
Logan Pearsall Smith

I shall return.
Douglas MacArthur
Outline

F.1 Introduction
F.2 Documentation Comments
F.3 Documenting Java Source Code
F.4 javadoc
F.5 Files Produced by javadoc

Summary • Terminology

F.1 Introduction

In this appendix, we provide an introduction to the Java 2 Software Development Kit’s javadoc utility for creating HTML files that document Java code. This is the tool used by Sun Microsystems to create the Java API documentation (Fig. F.1). We discuss the special Java comments and tags required by javadoc to create documentation based on your source code and how to execute the javadoc tool.

For detailed information on javadoc, visit the javadoc home page at

java.sun.com/j2se/1.3/docs/tooldocs/javadoc/index.html

F.2 Documentation Comments

Before HTML files can be generated with the javadoc tool, programmers must insert special comments—called documentation comments—into their source files. Documentation comments are the only comments recognized by javadoc. Documentation comments begin with /** and end with */. An example of a simple documentation comment is

/** Sorts integer array using MySort algorithm */

Like other comments, documentation comments are not translated into bytecodes. Because javadoc is used to create HTML files, documentation comments can contain HTML tags. For example, the documentation comment

/** Sorts integer array using \textit{MySort} algorithm */

which contains the HTML emphasis tags \textit{ and }\textit{ is valid. In the generated HTML files, MySort will appear as emphasized text (normally italic). As we will see, javadoc tags can be inserted into the documentation comments to help javadoc document your source code. These tags—which begin with an @ symbol—are not HTML tags.

F.3 Documenting Java Source Code

Figure F.2 presents a modified version of the Time3 class from Fig. 8.8 that contains documentation comments. In the text that follows the example, we thoroughly discuss each of the javadoc tags used in the documentation comments. We discuss how to use the javadoc tool to generate HTML documentation from this file in Section F.4.
Appendix F

Creating HTML Documentation with javadoc (on CD)

Fig. F.1  Java API documentation.

```java
// Fig. F.2: Time3.java
// Time3 class definition with set and get methods
package com.deitel.jhtp4.appenF;

// Java core packages
import java.text.DecimalFormat;

/**
 * This class maintains the time in 24-hour format.
 * @see java.lang.Object
 * @author Deitel & Associates, Inc.
 */
public class Time3 extends Object {
  private int hour;     // 0 - 23

  public Time3() {
  }

  public void setHour(int hour) {
    this.hour = hour;
  }

  public int getHour() {
    return hour;
  }

  public static void main(String[] args) {
    Time3 time = new Time3();
    time.setHour(15);
    System.out.println("The time is "+ time.getHour() + ":00.");
  }
}
```

Fig. F.2  A Java source code file containing documentation comments (part 1 of 5).
private int minute;  // 0 - 59
private int second;  // 0 - 59

/**
 * Time3 constructor initializes each instance variable
 * to zero. Ensures that Time object starts in a
 * consistent state.
 * @throws <code>Exception</code> in the case of an invalid time
 */
public Time3() throws Exception
{
    setTime( 0, 0, 0 );
}

/**
 * Time3 constructor: hour supplied, minute and second
 * defaulted to 0
 * @param h the hour
 * @throws <code>Exception</code> in the case of an invalid time
 */
public Time3( int h ) throws Exception
{
    setTime( h, 0, 0 );
}

/**
 * Time3 constructor: hour and minute supplied, second
 * defaulted to 0
 * @param h the hour
 * @param m the minute
 * @throws <code>Exception</code> in the case of an invalid time
 */
public Time3( int h, int m ) throws Exception
{
    setTime( h, m, 0 );
}

/**
 * Time3 constructor: hour, minute and second supplied
 * @param h the hour
 * @param m the minute
 * @param s the second
 * @throws <code>Exception</code> in the case of an invalid time
 */
public Time3( int h, int m, int s ) throws Exception
{
    setTime( h, m, s );
}
/**
 * Time3 constructor: another Time3 object supplied
 * @param time Time3 object
 * @throws <code>Exception</code> in the case of an invalid time
 */

public Time3( Time3 time ) throws Exception
{
    setTime( time.getHour(), time.getMinute(),
             time.getSecond() );
}

// Set Methods
/**
 * Set a new time value using universal time. Perform
 * validity checks on data. Set invalid values to zero.
 * @param h the hour
 * @param m the minute
 * @param s the second
 * @see com.deitel.jhtp4.appendF.Time3#setHour
 * @see Time3#setMinute
 * @see #setSecond
 * @throws <code>Exception</code> in the case of an invalid time
 */

public void setTime( int h, int m, int s ) throws Exception
{
    setHour( h );    // set the hour
    setMinute( m );  // set the minute
    setSecond( s );  // set the second
}

/**
 * Sets the hour
 * @param h the hour
 * @throws Exception in the case of an invalid time
 */

public void setHour( int h ) throws Exception
{
    if ( h >= 0 && h < 24 )
        hour = h;
    else
        throw new Exception();
}

/**
 * Sets the minute
 * @param m the minute
 * @throws Exception in the case of an invalid time
 */

public void setMinute( int m ) throws Exception
{
    if ( m >= 0 && m < 60 )
        minute = m;
```java
else
    throw new Exception();
}
/**
 * Sets the second
 * @param m the minute
 * @throws Exception in the case of an invalid time
 */
public void setSecond(int s) throws Exception
{
    if (s >= 0 && s < 60)
        second = s;
    else
        throw new Exception();
}
// Get Methods
/**
 * Gets the hour
 * @return an <code>int</code> specifying the hour.
 */
public int getHour()
{
    return hour;
}
/**
 * Gets the minute
 * @return an <code>int</code> specifying the minute.
 */
public int getMinute()
{
    return minute;
}
/**
 * Gets the second
 * @return an <code>int</code> specifying the second.
 */
public int getSecond()
{
    return second;
}
/**
 * Convert to <code>String</code> in universal-time format
 * @return a <code>String</code> representation
 * of the time in universal-time format
 */
public String toUniversalString()
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );
```

Fig. F.2 A Java source code file containing documentation comments (part 4 of 5).
Appendix F  Creating HTML Documentation with javadoc (on CD)  1375

Documentation comments are placed on the line before a class definition, an interface
definition, a constructor, a method and a field (i.e., an instance variable or a reference). The
first documentation comment (lines 8–12) introduces class Time3. The line

* This class maintains the time in 24-hour format.

is a description of class Time3 provided by the programmer. The description can contain
as many lines as necessary to provide a description of the class to any programmer who may
use it. Tags @see and @author are used to specify a See Also: note and an Author: note, respectively in the HTML documentation (Fig. F.3). The See Also: note specifies other related classes that may be of interest to a programmer using this class. The @author tag specifies the author of the class. More than one @author tag can be used to
document multiple authors. Note that the asterisks (*) on each line between /** and */
are not required. This is a convention used by programmers to align descriptions and javadoc tags. When parsing a documentation comment, javadoc discards all whitespace
characters up to the first non-whitespace character in each line. If the first non-whitespace
character encountered is an asterisk, it is also discarded.

Notice that the documentation comment in lines 8–12 immediately precedes the class
definition—any code placed between the documentation comment and the class definition
causes javadoc to ignore the documentation comment. This is also true of other code
structures (e.g., constructors, methods, instance variables, etc.).

Common Programming Error F.1

Placing an import statement between the class comment and the class declaration is a log-
ic error. This causes the class comment to be ignored by javadoc.
Defining several fields in one comma-separated statement with a single comment above that statement, will result in \textit{javadoc} using that comment for all of the fields.

The documentation comment on lines 32–37 describes one of the \texttt{Time3} constructors. The tag \texttt{@param} describes a parameter to the method. Parameters appear in the HTML document in a \textit{Parameters:} note (Fig. F.4) that is followed by a list of all parameters specified with the \texttt{@param} tag. For this constructor, the parameter’s name is \texttt{h} and its description is “the hour.” Tag \texttt{@param} can be used only with methods and constructors.

The tag \texttt{@throws} specifies the exceptions thrown by this method. Like \texttt{@param} tags, \texttt{@throws} tags are only used with methods and constructors. One \texttt{@throws} should be supplied for each type of exception thrown by the method.

To produce proper \textit{javadoc} documentation, you must declare every instance variable on a separate line.

Documentation comments can contain multiple \texttt{@param} and \texttt{@see} tags. The documentation comment at lines 79–89 describes method \texttt{setTime}. The HTML generated for this method is shown in Fig. F.5. Three \texttt{@param} tags describe the method’s parameters. This results in one \textit{Parameters:} note which lists the three parameters. Methods \texttt{setHour}, \texttt{setMinute} and \texttt{setSecond} are tagged with \texttt{@see} to create hyperlinks to their descriptions in the HTML document. A \# character is used instead of a dot when tagging a method or a field. This creates a link to the name that follows the \# character. We demon-
strate three different ways (i.e., the fully qualified name, class name qualification and no qualification) to tag methods using \texttt{@see} on lines 85–87. If the fully qualified name is not given (as in lines 86–87), \texttt{javadoc} looks for the specified method or field in the following order: current class, superclasses, package and imported files.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figF4.png}
\caption{The Parameters: note generated by javadoc.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figF5.png}
\caption{HTML documentation for method \texttt{setTime}.}
\end{figure}
The only other tag used in this file is \texttt{@return} which specifies a \textbf{Returns:} note in the HTML documentation (Fig. F.6). The comment at lines 137–140 documents method \texttt{getHour}. Tag \texttt{@return} describes a method’s return type to help the programmer understand how to use the return value of the method. By \texttt{javadoc} convention, programmers typeset source code (i.e., keywords, identifiers, expressions, etc.) with the HTML tags \texttt{<code>} and \texttt{</code>}.

\textbf{Good Programming Practice F.1} \\
Changing source code fonts in \texttt{javadoc} tags helps code names stand out from the rest of the description.

In addition to the tags presented in this example, \texttt{javadoc} recognizes four other tags. Figure F.7 summarizes the most commonly used tags.

<table>
<thead>
<tr>
<th>\textbf{javadoc tag}</th>
<th>\textbf{Description}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{@author}</td>
<td>Adds a Author: note if the \texttt{-author} option is used when executing \texttt{javadoc}.</td>
</tr>
<tr>
<td>\texttt{@param}</td>
<td>Used to describe the parameters of methods and constructors.</td>
</tr>
<tr>
<td>\texttt{@return}</td>
<td>Adds a Returns: note describing a methods return type.</td>
</tr>
<tr>
<td>\texttt{@see}</td>
<td>Adds a See Also: note containing hyperlinks to related classes or methods.</td>
</tr>
<tr>
<td>\texttt{@throws}</td>
<td>Adds a Throws: note that specifies the exceptions thrown by the method. \texttt{@exception} is a synonym for \texttt{@throws}.</td>
</tr>
<tr>
<td>\texttt{@deprecated}</td>
<td>Adds a Deprecated note. These are notes to programmers indicating that they should not use the specified features of the class. \texttt{Deprecated} notes normally appear when a class has been enhanced with new and improved features, but older features are maintained for backwards compatibility.</td>
</tr>
</tbody>
</table>

\textbf{Fig. F.6} \quad HTML documentation for method \texttt{getHour}.

\textbf{Fig. F.7} \quad Common \texttt{javadoc} tags (part 1 of 2).
In this section, we discuss how to execute the javadoc tool on a Java source file to create HTML documentation for the class in the file. Like other tools, javadoc is executed from the command line. The general form of the javadoc command is:

```
javadoc options packages sources @files
```

where `options` is a list of command line options, `packages` is a list of packages the user would like to document, `sources` is a list of java source files to document and `@files` is a text file containing the names of `packages` and/or `source files` to send to the `javadoc` utility, so that it can create documentation for those packages and classes. The wildcard character `*` can be used to specify multiple sources. (e.g., `c:\*.java`). [Note: All items are separated by spaces and `@files` is one word.] Figure F.8 shows a DOS window containing the javadoc command we typed to generate the HTML documentation.

In Fig. F.8, the `-d` argument specifies the directory (e.g., `c:\docs`) where the HTML files will be stored on disk. We use the `-link` option so that our documentation links to Sun’s documentation. This creates a hyperlink between our documentation and Sun’s documentation (see Fig. F.5 where Java class `Exception` from package `java.lang` is hyperlinked). Without the `-link` argument, Exception appears as text in the HTML document—not a hyperlink. The `-author` argument instructs javadoc to process the `@author` tag (it ignores this tag by default).

### F.5 Files Produced by javadoc

In the last section, we executed the javadoc tool on the `Time3.java` file. When javadoc executes, it displays the name of each HTML file it creates (see Fig. F.8). From the source file, javadoc created an HTML document named `Time3.html` for the class. If the source file contained multiple classes or interfaces, a separate HTML document is created for each class. Because class `Time3` belongs to a package, the page is created in the directory `C:\docs\com\deitel\jhtp4\appenF` (on Win32 platforms). The `c:\docs` directory was specified with the `-d` command line option of javadoc, and the remaining directories were created based on the `package` statement.
Another file that `javadoc` creates is `index.html` the starting HTML page in the documentation. To view the documentation you generate with `javadoc`, load `index.html` into your web browser. In Fig. F.9, the right frame contains the page `index.html` and the left frame contains the page `allclasses-frame.html` which contains links to the source code’s classes. [Note: Because our example does not contain multiple packages, there is no frame listing the packages. Normally this frame would appear above the left frame (containing “All Classes”) as in Fig. F.1.]

The navigation bar (at the top of the right frame in Fig. F.9) indicates which HTML page is currently loaded by highlighting the page’s link (e.g., the Class link in Fig. F.9).

Clicking the Tree link (Fig. F.10) displays a class hierarchy for all the classes displayed in the left frame. In our example, we documented only class `Time3`—which extends Object. Clicking the Deprecated link loads deprecated-list.html into the right frame. This page contains a list of all deprecated names. Because we did not use the @deprecated tag in this example, this page does not contain any information. Clicking the Index link loads the `index-all.html` page which contains an alphabetical list of all classes, interfaces, methods and fields.

Figure F.11 shows class Time3’s index-all.html page loaded into a Web browser. Clicking the Help link loads `helpdoc.html` (Fig. F.12). This is a help file for navigating the documentation. A default help file is provided, however the programmer can specify other help files.

Among the other files generated by `javadoc` are `serialized-form.html` which documents Serializable and Externalizable classes and `package-list.txt` which is used by the -link command-line argument and is not actually part of the documentation.

---

**Fig. F.8** Using the `javadoc` tool.
Fig. F.9  Class Time3's index.html.

Fig. F.10  Tree page.
Fig. F.11  Time3’s index-all.html page.

Fig. F.12  Time3’s helpdoc.html page.
**TERMINOLOGY**

- `allclasses-frame.html`: argument
- `author` tag
- `@author javadoc` tag
- `Class` item in the navigator bar
- `-d` argument
- `deprecated-list.html`
- `@deprecated` tag
- documentation comment
- `Exception` note
- `@exception` tag
- `Help`
- `helpdoc.html`
- `index.html`
- `index-all.html`
- `javadoc`
- `-link` argument

- `@link` tag
- `name-frame.html`
- `Overrides`: note
- `Parameters`: note
- `@param` tag
- `Returns`: note
- `@return` tag
- `See Also`: note
- `@see` tag
- `serialized-form.html`
- `@serial` tag
- `@serialData` tag
- `@since` tag
- `@throws` tag
- `Tree` link
- `@version` tag
Elevator Events and Listener Interfaces (on CD)

G.1 Introduction
In “Thinking About Objects” Section 10.22, we discussed how event handling works in our elevator simulation. We mentioned that for an object to receive an event, that object must register as a listener for that event. Therefore, the class of that object must implement an appropriate listener interface that contains methods that receive an event object as a parameter. In this section, we present the events and listener interfaces used in our simulation.

G.2 Events
The next eight figures (Fig. G.1–Fig. G.7) contain the events of the system. Each event inherits from class ElevatorModelEvent in Fig. G.1. This class contains a Location reference (line 11) that represent where the event was generated—in our simulation, this reference is the Elevator object or either Floor object. Class ElevatorModelEvent also contains an Object reference (line 14) representing the source object that generated the event. Methods getLocation (lines 30–33) and getSource (lines 42–45) return the Location and Object references, respectively. Note that each subclass of ElevatorModelEvent (Fig. G.2–Fig. G.7) provides only a constructor calling the constructor of class ElevatorModelEvent. As we mentioned in Section 10.22, dividing class ElevatorModelEvent into several subclass events makes event handling easier to understand in our simulation.

```java
1    // ElevatorModelEvent.java
2    // Basic event packet in Elevator simulation
3    package com.deitel.jhtp4.elevator.event;
```

Fig. G.1 ElevatorModelEvent superclass for events in the elevator simulation model (part 1 of 2).
import com.deitel.jhtp4.elevator.model.*;

public class ElevatorModelEvent {

   // Location where ElevatorModelEvent was generated
   private Location location;

   // source Object that generated ElevatorModelEvent
   private Object source;

   // ElevatorModelEvent constructor sets Location
   public ElevatorModelEvent( Object source, Location location )
   {
      setSource( source );
      setLocation( location );
   }

   // set ElevatorModelEvent Location
   public void setLocation( Location eventLocation )
   {
      location = eventLocation;
   }

   // get ElevatorModelEvent Location
   public Location getLocation()
   {
      return location;
   }

   // set ElevatorModelEvent source
   private void setSource( Object eventSource )
   {
      source = eventSource;
   }

   // get ElevatorModelEvent source
   public Object getSource()
   {
      return source;
   }
}

Fig. G.1 ElevatorModelEvent superclass for events in the elevator simulation model (part 2 of 2).

// BellEvent.java
// Indicates that Bell has rung
package com.deitel.jhtp4.elevator.event;

Fig. G.2 BellEvent ElevatorModelEvent subclass indicating that the Bell has rung (part 1 of 2).
import com.deitel.jhtp4.elevator.model.*;

public class BellEvent extends ElevatorModelEvent {
    // BellEvent constructor
    public BellEvent( Object source, Location location ) {
        super( source, location );
    }
}

Fig. G.2 BellEvent ElevatorModelEvent subclass indicating that the Bell has rung (part 2 of 2).

import com.deitel.jhtp4.elevator.model.*;

public class ButtonEvent extends ElevatorModelEvent {
    // ButtonEvent constructor
    public ButtonEvent( Object source, Location location ) {
        super( source, location );
    }
}

Fig. G.3 ButtonEvent ElevatorModelEvent subclass indicating that a Button has changed state.

import com.deitel.jhtp4.elevator.model.*;

public class DoorEvent extends ElevatorModelEvent {
    // DoorEvent constructor
    public DoorEvent( Object source, Location location ) {
        super( source, location );
    }
}

Fig. G.4 DoorEvent ElevatorModelEvent subclass indicating that a Door has changed state.
Class \texttt{PersonMoveEvent} (Fig. G.7) has a slightly different structure than that of the other event classes. Line 11 declares \texttt{int} attribute \texttt{ID}. We will discover in Appendix I that the \texttt{ElevatorView} obtains this attribute through method \texttt{getID} (lines 22–25) to determine which \texttt{Person} sent the event.
G.3 Listeners

The next eight figures (Fig. G.8–Fig. G.14) contain the listener interfaces for the elevator simulation. BellListener (Fig. G.8) provides method bellRang (lines 8), which is invoked when the Bell has rung. ButtonListener (Fig. G.9) provides methods buttonPressed (line 8) and buttonReset (line 11), which listen when a Button is pressed or reset. DoorListener (Fig. G.10) provides methods doorOpened (line 8) and doorClosed (line 11), which listen for a Door opening or closing. ElevatorMoveListener (Fig. G.11) provides methods elevatorDeparted (line 8) and elevatorArrived (line 11), which listen for Elevator departures and arrivals. LightListener (Fig. G.12) provides methods lightTurnedOn (line 8) and lightTurnedOff (line 11) that listen for Light state changes. PersonMoveListener (Fig. G.13) provides methods personCreated (line 8), personArrived (line 11), personDeparted (line 14), personPressedButton (line 17–18), personEntered (line 21) and personExited (line 24). These methods listen for when a Person has been created, has arrived at or departed from the Elevator, pressed a Button, entered the Elevator, or exited the simulation, respectively. Lastly, ElevatorModelListener (Fig. G.14) inherits behaviors from all listener interfaces. The ElevatorView uses interface ElevatorModelListener in Section 13.17 and Appendix I to receive events from the ElevatorModel.
public interface BellListener {
    // invoked when Bell has rungs
    public void bellRang(BellEvent bellEvent);
}

// ButtonListener.java
// Methods invoked when Button has been either pressed or reset
package com.deitel.jhtp4.elevator.event;

public interface ButtonListener {
    // invoked when Button has been pressed
    public void buttonPressed(ButtonEvent buttonEvent);
    // invoked when Button has been reset
    public void buttonReset(ButtonEvent buttonEvent);
}

// DoorListener.java
// Methods invoked when Door has either opened or closed
package com.deitel.jhtp4.elevator.event;

public interface DoorListener {
    // invoked when Door has opened
    public void doorOpened(DoorEvent doorEvent);
    // invoked when Door has closed
    public void doorClosed(DoorEvent doorEvent);
}

// ElevatorMoveListener.java
// Methods invoked when Elevator has either departed or arrived
package com.deitel.jhtp4.elevator.event;

public interface ElevatorMoveListener {
    // invoked when Elevator has departed
    public void elevatorDeparted(ElevatorMoveEvent moveEvent);
}

// Figure G.8 Interface BellListener method when Bell has rung (part 2 of 2).
// Figure G.9 Interface ButtonListener methods when Button has been either pressed or reset.
// Figure G.10 Interface DoorListener methods when Door has either opened or closed.
// Figure G.11 Interface ElevatorMoveListener methods when Elevator has either departed from or arrived on a Floor (part 1 of 2).
// invoked when Elevator has arrived
public void elevatorArrived(ElevatorMoveEvent moveEvent);
}

Fig. G.11 Interface ElevatorMoveListener methods when Elevator has either departed from or arrived on a Floor (part 2 of 2).

// LightListener.java
// Methods invoked when Light has either turned on or off
package com.deitel.jhtp4.elevator.event;

public interface LightListener {
    // invoked when Light has turned on
    public void lightTurnedOn(LightEvent lightEvent);
    // invoked when Light has turned off
    public void lightTurnedOff(LightEvent lightEvent);
}

Fig. G.12 Interface LightListener method for when Light has either turned on or off.

// PersonMoveListener.java
// Methods invoked when Person moved
package com.deitel.jhtp4.elevator.event;

public interface PersonMoveListener {
    // invoked when Person has been instantiated in model
    public void personCreated(PersonMoveEvent moveEvent);
    // invoked when Person arrived at elevator
    public void personArrived(PersonMoveEvent moveEvent);
    // invoked when Person departed from elevator
    public void personDeparted(PersonMoveEvent moveEvent);
    // invoked when Person pressed Button
    public void personPressedButton(PersonMoveEvent moveEvent);
    // invoked when Person entered Elevator
    public void personEntered(PersonMoveEvent moveEvent);
    // invoked when Person exited simulation
    public void personExited(PersonMoveEvent moveEvent);
}

Fig. G.13 Interface PersonMoveListener methods when Person has moved.
G.4 Component Diagrams Revisited

In Section 13.17, we introduced the component diagram for the elevator simulation. In our simulation, the `ElevatorView` and every object in the model import package `event`. Figure G.15 presents the component diagram for package `event`. Each component in package `event` maps to a class from Fig. G.1–Fig. G.14. According to the component diagram, `ElevatorView.java` of package `view` aggregates package `event`. In Java, this aggregation translates to class `ElevatorView` importing package `event`. Also according to Fig. G.15, package `model` aggregates package `event`—i.e., each component in package `model` contains an aggregation with all components in package `event`. (We show all components of package `model` in a separate component diagram in Appendix H.) In Java, this aggregation translates to each class in package `model` that imports `package event`.

This concludes the appendix on the events and listener interfaces of the elevator simulation. We hope you have found it a useful reference for the material on event handling discussed in “Thinking About Objects” Section 10.22. In the next two appendices, we implement the design for the model and the view, and we provide the component diagrams for packages `model` and `view`.

```java
1 // ElevatorModelListener.java
2 // Listener for ElevatorView from ElevatorModel
3 package com.deitel.jhtp4.elevator.event;
4
5 // ElevatorModelListener inherits all Listener interfaces
6 public interface ElevatorModelListener extends BellListener,
7 ButtonListener, DoorListener, ElevatorMoveListener,
8 LightListener, PersonMoveListener {
9 }
```

Fig. G.14 Interface `ElevatorModelListener` allows the model to send all events to the view.
Fig. G.15 Component diagram for package **event**.
H.1 Introduction

After reading the “Thinking About Objects” sections, you should have a comfortable grasp of the design process and of the UML diagrams that pertain to our simulation. This appendix presents the code for all 10 classes that collectively represent the model, and concludes the discussion of its workings. We discuss each class separately and in detail.

H.2 Class ElevatorModel

As discussed in Section 13.17, class ElevatorModel (Fig. H.1) ties together the objects that comprise the elevator simulation model. The ElevatorModel serves to send events from the model to the view. The ElevatorModel also instantiates new Persons and allows each Floor to obtain a reference to the ElevatorShaft.

```java
// ElevatorModel.java
// Elevator simulation model with ElevatorShaft and two Floors
package com.deitel.jhttp4.elevator.model;

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

// Deitel packages
import com.deitel.jhttp4.elevator.event.*;
import com.deitel.jhttp4.elevator.ElevatorConstants;

public class ElevatorModel implements ElevatorModelListener,
    ElevatorConstants {

```

Fig. H.1 Class ElevatorModel represents the model in our elevator simulation (part 1 of 7).
// declare two-Floor architecture in simulation
private Floor firstFloor;
private Floor secondFloor;

// ElevatorShaft in simulation
private ElevatorShaft elevatorShaft;

// objects listening for events from ElevatorModel
private Set personMoveListeners;
private DoorListener doorListener;
private ButtonListener buttonListener;
private LightListener lightListener;
private BellListener bellListener;
private ElevatorMoveListener elevatorMoveListener;

// cumulative number of people in simulation
private int numberOfPeople = 0;

// constructor instantiates ElevatorShaft and Floors
public ElevatorModel()
{
    // instantiate firstFloor and secondFloor objects
    firstFloor = new Floor( FIRST_FLOOR_NAME );
    secondFloor = new Floor( SECOND_FLOOR_NAME );

    // instantiate ElevatorShaft object
    elevatorShaft =
        new ElevatorShaft( firstFloor, secondFloor );

    // give elevatorShaft reference to first and second Floor
    firstFloor.setElevatorShaft( elevatorShaft );
    secondFloor.setElevatorShaft( elevatorShaft );

    // register for events from ElevatorShaft
    elevatorShaft.setDoorListener( this );
    elevatorShaft.setButtonListener( this );
    elevatorShaft.addElevatorMoveListener( this );
    elevatorShaft.setLightListener( this );
    elevatorShaft.setBellListener( this );

    // instantiate Set for ElevatorMoveListener objects
    personMoveListeners = new HashSet( 1 );
}

// return Floor with given name
private Floor getFloor( String name )
{
    if ( name.equals( FIRST_FLOOR_NAME ) )
        return firstFloor;
    else

if ( name.equals( SECOND_FLOOR_NAME ) )
    return secondFloor;
else
    return null;

} // end method getFloor

// add Person to Elevator Simulator
public void addPerson( String floorName )
{
    // instantiate new Person and place on Floor
    Person person =
        new Person( numberOfPeople, getFloor( floorName ) );
    person.setName( Integer.toString( numberOfPeople ) );

    // register listener for Person events
    person.setPersonMoveListener( this );

    // start Person thread
    person.start();

    // increment number of Person objects in simulation
    numberOfPeople++;

} // end method addPerson

// invoked when Elevator has departed from Floor
public void elevatorDeparted( ElevatorMoveEvent moveEvent )
{
    elevatorMoveListener.elevatorDeparted( moveEvent );
}

// invoked when Elevator has arrived at destination Floor
public void elevatorArrived( ElevatorMoveEvent moveEvent )
{
    elevatorMoveListener.elevatorArrived( moveEvent );
}

// send PersonMoveEvent to listener, depending on event type
private void sendPersonMoveEvent(
    int eventType, PersonMoveEvent event )
{
    Iterator iterator = personMoveListeners.iterator();
    while ( iterator.hasNext() ) {
        PersonMoveListener listener =
            ( PersonMoveListener ) iterator.next();

        listener.onPersonMoveEvent( event );
    }
}

Fig. H.1 Class ElevatorModel represents the model in our elevator simulation (part 3 of 7).
// send Event to this listener, depending on eventType
switch ( eventType ) {

    // Person has been created
    case Person.PERSON_CREATED:
        listener.personCreated( event );
        break;

    // Person arrived at Elevator
    case Person.PERSON_ARRIVED:
        listener.personArrived( event );
        break;

    // Person entered Elevator
    case Person.PERSON_ENTERING_ELEVATOR:
        listener.personEntered( event );
        break;

    // Person pressed Button object
    case Person.PERSON_PRESSING_BUTTON:
        listener.personPressedButton( event );
        break;

    // Person exited Elevator
    case Person.PERSON_EXITING_ELEVATOR:
        listener.personDeparted( event );
        break;

    // Person exited simulation
    case Person.PERSON_EXITED:
        listener.personExited( event );
        break;

    default:
        break;
}
} // end method sendPersonMoveEvent

// invoked when Person has been created in model
public void personCreated( PersonMoveEvent moveEvent )
{
    sendPersonMoveEvent( Person.PERSON_CREATED, moveEvent );
}

// invoked when Person has arrived at Floor's Button
public void personArrived( PersonMoveEvent moveEvent )
{
    sendPersonMoveEvent( Person.PERSON_ARRIVED, moveEvent );
}

Fig. H.1 Class ElevatorModel represents the model in our elevator simulation (part 4 of 7).
Fig. H.1 Class ElevatorModel represents the model in our elevator simulation (part 5 of 7).
// invoked when Bell has rung
public void bellRang( BellEvent bellEvent )
{
    bellListener.bellRang( bellEvent );
}

// invoked when Light has turned on
public void lightTurnedOn( LightEvent lightEvent )
{
    lightListener.lightTurnedOn( lightEvent );
}

// invoked when Light has turned off
public void lightTurnedOff( LightEvent lightEvent )
{
    lightListener.lightTurnedOff( lightEvent );
}

// set listener for ElevatorModelListener
public void setElevatorModelListener( ElevatorModelListener listener )
{
    // ElevatorModelListener extends all interfaces below
    addPersonMoveListener( listener );
    setElevatorMoveListener( listener );
    setDoorListener( listener );
    setButtonListener( listener );
    setLightListener( listener );
    setBellListener( listener );
}

// set listener for PersonMoveEvents
public void addPersonMoveListener( PersonMoveListener listener )
{
    personMoveListeners.add( listener );
}

// set listener for DoorEvents
public void setDoorListener( DoorListener listener )
{
    doorListener = listener;
}

// set listener for ButtonEvents
public void setButtonListener( ButtonListener listener )
{
    buttonListener = listener;
}

Fig. H.1 Class ElevatorModel represents the model in our elevator simulation (part 6 of 7).
The class diagram in Fig. 15.21 shows that class ElevatorModel contains one instance of class ElevatorShaft and two instances of class Floor, so ElevatorModel declares object elevatorShaft (line 20) and objects firstFloor and secondFloor (lines 16–17). Lines 37–46 instantiate these objects and give each Floor object a reference to the ElevatorShaft object. Fig. 15.21 also shows that class ElevatorModel creates Person objects. According to Fig. 15.21, class ElevatorModel contains method addPerson (lines 75–91), which creates and places a Person on the specified Floor. Line 86 of method addPerson starts the Person’s thread and line 89 increments the cumulative number of Person objects in the simulation.

As previously mentioned, class ElevatorModel sends events from the model to the view. The class declaration (lines 12–13) and lines 49–53 reveal that the ElevatorModel listens for several types of events from the ElevatorShaft—this is how the ElevatorModel receives events from the objects that comprise the model. Specifically, class ElevatorModel implements interface ElevatorModelListener, which implements all interfaces in package event. Lines 23–28 declare the listener objects to which the ElevatorModel sends the events it receives from the ElevatorShaft. As mentioned in Section 13.17, the ElevatorFrame (the application) registers the ElevatorView as a listener for events from the ElevatorModel—this is how the ElevatorModel sends events from the model to the view.

Lines 94–236 of class ElevatorModel implement all methods of interface ElevatorModelListener, and lines 239–288 provide addListener methods to register a listener (in this case, the listener is the ElevatorView) for all events. In fact, two-thirds of the class devotes itself to “bubbling up” messages from the model to the view.

We presented a class diagram that showed the realizations of the elevator model in Fig. 11.27. We alter this diagram to accommodate the fact that class ElevatorModel implements all interfaces through interface ElevatorModelListener. In addition, class ElevatorShaft must implement more interfaces to receive events from class...
**Elevator** so that **ElevatorShaft** can send these events to the **ElevatorModel**. We present the class diagram showing all realizations for the model in Fig. H.2 and Fig. H.3—Fig. H.2 shows the relationship between the classes in the model and the listener interfaces, and Fig. H.3 shows the relationship between the listener interfaces and interface **ElevatorModelListener** (we could have created one class diagram showing both types of relationships, but the diagram would have been too cluttered).

Figure H.4 describes the contents of the class diagram that shows elevator model realizations. The substantial changes are that class **ElevatorShaft** now implements interface **ElevatorMoveListener**, and **ElevatorModel** implements interface **ElevatorModelListener**, which implements all interfaces.
H.3 Classes Location and Floor

We need to represent the location of the Person in the simulation. The Person could have an int attribute describing on which Floor the Person is walking; however, the Person does not occupy either Floor when riding the Elevator. As described in “Thinking About Objects” Section 9.23, our solution is for the Person to maintain a Location reference, which references either a Floor or the Elevator, depending on the whereabouts of the Person. To implement this feature, classes Floor and Elevator extend abstract superclass Location (Fig. H.5).

<table>
<thead>
<tr>
<th>Class</th>
<th>implements Listener</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElevatorModel</td>
<td>ElevatorModelListener</td>
</tr>
<tr>
<td>ElevatorModelListener</td>
<td>PersonMoveListener</td>
</tr>
<tr>
<td></td>
<td>ElevatorMoveListener</td>
</tr>
<tr>
<td></td>
<td>ButtonListener</td>
</tr>
<tr>
<td></td>
<td>DoorListener</td>
</tr>
<tr>
<td></td>
<td>BellListener</td>
</tr>
<tr>
<td></td>
<td>LightListener</td>
</tr>
<tr>
<td>Door, Light, Bell, Button</td>
<td>ElevatorMoveListener</td>
</tr>
<tr>
<td>Elevator</td>
<td>ButtonListener</td>
</tr>
<tr>
<td></td>
<td>DoorListener</td>
</tr>
<tr>
<td></td>
<td>BellListener</td>
</tr>
<tr>
<td>ElevatorShaft</td>
<td>LightListener</td>
</tr>
<tr>
<td></td>
<td>ButtonListener</td>
</tr>
<tr>
<td></td>
<td>DoorListener</td>
</tr>
<tr>
<td></td>
<td>BellListener</td>
</tr>
<tr>
<td></td>
<td>ElevatorMoveListener</td>
</tr>
<tr>
<td>Person</td>
<td>DoorListener</td>
</tr>
</tbody>
</table>
Location contains String locationName (line 11), which may contain either value firstFloor, secondFloor or elevator to describe the three locations the Person may occupy. Lines 26 and 29 declare abstract methods getButton and getDoor, respectively. Using these methods, a Floor returns references to objects associated with that Floor, and the Elevator returns references to the objects associated with the Elevator. The Location reference allows a Person to press a Button and to know when a Door has opened. For example, if we wish a Person to press a Button, we write

```
person.getLocation().getButton().pressButton();
```

Therefore, our use of an abstract superclass provides an alternate means for objects in our model to interact. Figure H.6 presents class Floor, a subclass of class Location. The Floor constructor (lines 15–18) takes as a String argument the value firstFloor or secondFloor to identify the Floor. Line 17 invokes method setLocationName to assign the value of this String to attribute locationName, inherited from superclass Location.
Appendix H Elevator Model (on CD) 1403

// Floor.java
// Represents a Floor located next to an ElevatorShaft
package com.deitel.jhtp4.elevator.model;

// Deitel packages
import com.deitel.jhtp4.elevator.ElevatorConstants;

public class Floor extends Location
    implements ElevatorConstants {

    // reference to ElevatorShaft object
    private ElevatorShaft elevatorShaft;

    // Floor constructor sets name of Floor
    public Floor( String name ) {
        setLocationName( name );
    }

    // get first or second Floor Button, using Location name
    public Button getButton() {
        if ( getLocationName().equals( FIRST_FLOOR_NAME ) )
            return getElevatorShaft().getFirstFloorButton();
        else
            if ( getLocationName().equals( SECOND_FLOOR_NAME ) )
                return getElevatorShaft().getSecondFloorButton();
            else
                return null;
    }

    // get first or second Floor Door, using Location name
    public Door getDoor() {
        if ( getLocationName().equals( FIRST_FLOOR_NAME ) )
            return getElevatorShaft().getFirstFloorDoor();
        else
            if ( getLocationName().equals( SECOND_FLOOR_NAME ) )
                return getElevatorShaft().getSecondFloorDoor();
            else
                return null;
    }

    // get ElevatorShaft reference
    public ElevatorShaft getElevatorShaft() {
    }

Fig. H.6 Class Floor—a subclass of Location—represents a Floor across which a Person walks to the Elevator (part 1 of 2).
H.4 Class Door

The doors are an essential part of the model, because they inform a person when to enter and exit the elevator—without the doors, no person would ride the elevator. The collaboration diagrams of Fig. 7.20, Fig. 10.25 and Fig. 15.18 presented the collaborations among the doors, the elevator and the person. Now, we provide a walkthrough of class door (Fig. H.7).

```java
// Door.java
// Sends DoorEvents to DoorListeners when opened or closed
package com.deitel.jhtp4.elevator.model;

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

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;

public class Door implements ElevatorMoveListener {

    // represent whether Door is open or closed
    private boolean open = false;

    // time before Door closes automatically
    public static final int AUTOMATIC_CLOSE_DELAY = 3000;

    // Set of DoorListeners
    private Set doorListeners;
```

Fig. H.7 Class Door, which represents a Door in the model, informs listeners when a Door has opened or closed (part 1 of 4).
// location where Door opened or closed
private Location doorLocation;

// Door constructor instantiates Set for DoorListeners
public Door()
{
doorListeners = new HashSet( 1 );
}

// add Door listener
public void addDoorListener( DoorListener listener )
{
    // prevent other objects from modifying doorListeners
    synchronized( doorListeners )
    {
doorListeners.add( listener );
    }
}

// remove Door listener
public void removeDoorListener( DoorListener listener )
{
    // prevent other objects from modifying doorListeners
    synchronized( doorListeners )
    {
doorListeners.remove( listener );
    }
}

// open Door and send all listeners DoorEvent objects
public void openDoor( Location location )
{
    if ( !open ) {
        open = true;
        // obtain iterator from Set
        Iterator iterator;
        synchronized( doorListeners )
        {
            iterator = new HashSet( doorListeners ).iterator();
        }
        // get next DoorListener
        while ( iterator.hasNext() ) {
            DoorListener doorListener = ( DoorListener ) iterator.next();
            // send doorOpened event to this DoorListener
            doorListener.doorOpened(
                new DoorEvent( this, location ) );
        }
    }

Fig. H.7 Class Door, which represents a Door in the model, informs listeners when a Door has opened or closed (part 2 of 4).
doorLocation = location;

// declare Thread that ensures automatic Door closing
Thread closeThread = new Thread(
    new Runnable() {
        public void run()
        {
            // close Door if open for more than 3 seconds
            try {
                Thread.sleep( AUTOMATIC_CLOSE_DELAY );
                closeDoor( doorLocation );
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        }
    } // end anonymous inner class

    closeThread.start();
}
} // end method openDoor

// close Door and send all listeners DoorEvent objects
public void closeDoor( Location location )
{
    if ( open ) {
        open = false;

        // obtain iterator from Set
        Iterator iterator;
        synchronized( doorListeners )
        {
            iterator = new HashSet( doorListeners ).iterator();
        }

        // get next DoorListener
        while ( iterator.hasNext() ) {
            DoorListener doorListener =
                ( DoorListener ) iterator.next();

            // send doorClosed event to this DoorListener
            doorListener.doorClosed(
                new DoorEvent( this, location ) );
        }
    } // end method closeDoor

Fig. H.7 Class Door, which represents a Door in the model, informs listeners when a Door has opened or closed (part 3 of 4).
Figure 15.12 indicates that class Venue contains boolean attribute open (line 14) to represent the state of the Venue (open or closed). Figure 15.12 also indicates that class Venue contains methods openVenue (lines 52–99) and closeVenue (lines 102–125). Method openVenue sends a doorOpened event to all registered VenueListeners (the Venue passes a DoorEvent object to the doorOpened method of each registered VenueListener), and method closeVenue sends a doorClosed event to all registered VenueListeners. Set venueListeners (line 20) stores all registered VenueListeners. A VenueListener wishing to receive DoorEvents from a Venue must invoke method addVenueListener (lines 32–39); those listeners that no longer wish to be a VenueListener for that Venue must invoke removeVenueListener (lines 42–49).

Line 56 of method openVenue opens the Venue by setting open to true. Lines 66–73 iterate Set venueListeners and send each object a doorOpened event. Lines 60–63 use a synchronized block obtaining the Iterator from Set venueListeners, because a Person, at any time, can add or remove itself from this Set. If method openVenue iterates venueListeners as a Person adds or removes itself from venueListeners, the JVM throws a ConcurrentModificationException—the Person is modifying the Set as method openVenue is iterating the same Set. Method openVenue avoids this situation with the synchronized block.

Method openVenue receives as an argument a Location reference on what Floor that Venue should open. Lines 71–72 send a DoorEvent using the Location reference to all registered VenueListeners. Method closeVenue sets open to false (thus closing the Venue) and invokes the closeVenue method on all registered VenueListeners.

We decided in Section 15.12 to make class Venue an active class, so that the Venue closes itself after three seconds of being open. Lines 78–95 of method openVenue instantiate a thread that handles this responsibility. Method run (lines 81–93) puts this thread to sleep for three seconds then closes the Venue. Line 97 of method openVenue starts the thread.

Lastly, according to Fig. H.2, Venue implements ElevatorMoveListener. In our simulation, the Elevator invokes method elevatorDeparted (line 134) when the
Elevator has departed and invokes method `elevatorArrived` (lines 137–140) when the Elevator has arrived. Method `elevatorArrived` calls method `openDoor`—the Door opens when the Elevator has arrived. Method `elevatorDeparted` does not perform any action. At a first glance, you may wonder why this method does not call method `closeDoor`. The reason is that a Door should close before the Elevator has departed so the passenger will not be injured—in our model, the Elevator calls method `closeDoor` before calling `elevatorDeparted`.

H.5 Class Button

Buttons (Fig. H.8) are important to the model as well, because they signal the Elevator to move between Floors. Figure 15.12 indicates that class Button contains boolean attribute `pressed` (line 14) to represent the state of the Button (pressed or reset). Figure 15.12 indicates also that class Button contains methods `pressButton` (lines 23–29) and `resetButton` (lines 32–38).

```java
// Button.java
// Sends ButtonEvents to ButtonListeners when accessed
package com.deitel.jhtp4.elevator.model;

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;

public class Button implements ElevatorMoveListener {

    // ButtonListener
    private ButtonListener buttonListener = null;

    // represent whether Button is pressed
    private boolean pressed = false;

    // set listener
    public void setButtonListener( ButtonListener listener )
    {
        buttonListener = listener;
    }

    // press Button and send ButtonEvent
    public void pressButton( Location location )
    {
        pressed = true;

        buttonListener.buttonPressed( new ButtonEvent( this, location ) );
    }

    // reset Button and send ButtonEvent
    public void resetButton( Location location )
    {
        pressed = false;
    }
}
```

Fig. H.8 Class Button, which represents a Button in the model, informs listeners when a Button has been pressed or reset (part 1 of 2).
Method `pressButton` sends a `buttonPressed` event to the registered `ButtonListener` (line 11), and method `resetButton` sends a `buttonReset` event to the `ButtonListener`. Method `setButtonListener` (lines 17–20) allows an object to receive `ButtonEvents` by registering itself as the `ButtonListener`.

Line 25 of method `pressButton` sets attribute `pressed` to `true`, and lines 27–28 pass a `ButtonEvent` to method `buttonPressed` of the `buttonListener`. Line 34 of method `resetButton` sets attribute `pressed` to `false` and line 36–37 pass a `ButtonEvent` to method `buttonReset` of the `buttonListener`.

Lastly, according to Fig. H.2, classes `Button` implements interface `ElevatorMoveListener`. Method `elevatorArrived` (lines 50–53) calls method `resetButton` to reset the `Button`.

### H.6 Class ElevatorShaft

Class `ElevatorShaft` (Fig. H.9) represents the `ElevatorShaft` in which the Elevator travels in the model. Most methods in class `ElevatorShaft` access `private` variables, listen for messages from the Elevator and send “bubble up” events to the `ElevatorModel`, which sends them to the `ElevatorView`. According to the class diagram of Fig. 15.21, class `ElevatorShaft` contains one `Elevator` object, two `Button` objects, two `Door` objects and two `Light` objects. The `Button`, `Door` and `Light` objects refer to the buttons, doors and lights on each `Floor`. Line 15 declares the `Elevator` object—`elevator`. Lines 18–19 declare the `Buttons firstFloorButton` and `secondFloorButton`. Lines 22–23 declare the `Doors firstFloorDoor` and `secondFloorDoor`. Lines 26–27 declare the `Lights firstFloorLight` and `secondFloorLight`. Lines 169–208 provide methods to access references to these objects.
// ElevatorShaft.java
// Represents elevator shaft, which contains elevator
package com.deitel.jhtp4.elevator.model;

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

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;

public class ElevatorShaft implements ElevatorMoveListener,
LightListener, BellListener {

    // Elevator
    private Elevator elevator;

    // Buttons on Floors
    private Button firstFloorButton;
    private Button secondFloorButton;

    // Doors on Floors
    private Door firstFloorDoor;
    private Door secondFloorDoor;

    // Lights on Floors
    private Light firstFloorLight;
    private Light secondFloorLight;

    // listeners
    private DoorListener doorListener;
    private ButtonListener buttonListener;
    private LightListener lightListener;
    private BellListener bellListener;
    private Set elevatorMoveListeners;

    // constructor initializes aggregated components
    public ElevatorShaft( Floor firstFloor, Floor secondFloor )
    {
        // instantiate Set for ElevatorMoveListeners
        elevatorMoveListeners = new HashSet( 1 );

        // anonymous inner class listens for ButtonEvents
        ButtonListener floorButtonListener =
            new ButtonListener()
            {
                // called when Floor Button has been pressed
                public void buttonPressed( ButtonEvent buttonEvent )
                {
                    // request elevator move to location
                    Location location = buttonEvent.getLocation();
                    buttonListener.buttonPressed( buttonEvent );
                } // end buttonPressed
            } // end anonymous inner class

        // request elevator move to firstFloor
        elevatorMoveListeners.add( floorButtonListener );
        elevatorMoveListeners.add( buttonListener );
        elevatorMoveListeners.add( lightListener );
        elevatorMoveListeners.add( bellListener );

        // request elevator move to secondFloor
        Location location = buttonEvent.getLocation();
        buttonListener.buttonPressed( buttonEvent );
    } // end constructor

} // end class

Fig. H.9 Class ElevatorShaft, which represents the ElevatorShaft, which
sends events from the Elevator to the ElevatorModel (part 1 of 6).
Appendix H

52        elevator.requestElevator( location );
53
54
55    // called when Floor Button has been reset
56    public void buttonReset( ButtonEvent buttonEvent )
57    {
58        buttonListener.buttonReset( buttonEvent );
59    }
60    } // end anonymous inner class
61
62    // instantiate Floor Buttons
63    firstFloorButton = new Button();
64    secondFloorButton = new Button();
65
66    // register anonymous ButtonListener with Floor Buttons
67    firstFloorButton.setButtonListener(
68        floorButtonListener );
69    secondFloorButton.setButtonListener(
70        floorButtonListener );
71
72    // Floor Buttons listen for ElevatorMoveEvents
73    addElevatorMoveListener( firstFloorButton );
74    addElevatorMoveListener( secondFloorButton );
75
76    // anonymous inner class listens for DoorEvents
77    DoorListener floorDoorListener = new DoorListener() { 
78
79        // called when Floor Door has opened
80        public void doorOpened( DoorEvent doorEvent )
81        {
82            // forward event to doorListener
83            doorListener.doorOpened( doorEvent );
84        }
85
86        // called when Floor Door has closed
87        public void doorClosed( DoorEvent doorEvent )
88        {
89            // forward event to doorListener
90            doorListener.doorClosed( doorEvent );
91        }
92    } // end anonymous inner class
93
94    // instantiate Floor Doors
95    firstFloorDoor = new Door();
96    secondFloorDoor = new Door();
97
98    // register anonymous DoorListener with Floor Doors
99    firstFloorDoor.addDoorListener( floorDoorListener );
100    secondFloorDoor.addDoorListener( floorDoorListener );
101
102    // instantiate Lights, then listen for LightEvents
103    firstFloorLight = new Light();

Fig. H.9 Class ElevatorShaft, which represents the ElevatorShaft, which sends events from the Elevator to the ElevatorModel (part 2 of 6).
```java
addElevatorMoveListener( firstFloorLight );
firstFloorLight.setLightListener( this );

secondFloorLight = new Light();
addElevatorMoveListener( secondFloorLight );
secondFloorLight.setLightListener( this );

// instantiate Elevator object
elevator = new Elevator( firstFloor, secondFloor );

// register for ElevatorMoveEvents from elevator
elevator.addElevatorMoveListener( this );

// listen for BellEvents from elevator
elevator.setBellListener( this );

// anonymous inner class listens for ButtonEvents from
elevator
elevator.setButtonListener(
    new ButtonListener() {
        // invoked when button has been pressed
        public void buttonPressed( ButtonEvent buttonEvent )
        {
            // send event to listener
            buttonListener.buttonPressed( buttonEvent );
        }

        // invoked when button has been reset
        public void buttonReset( ButtonEvent buttonEvent )
        {
            // send event to listener
            buttonListener.buttonReset( new ButtonEvent( this, elevator ) );
        }
    } // end anonymous inner class
);

// anonymous inner class listens for DoorEvents from
elevator
elevator.setDoorListener(
    new DoorListener() {
        // invoked when door has opened
        public void doorOpened( DoorEvent doorEvent )
        {
            // send event to listener
            doorListener.doorOpened( doorEvent );
        }
    } // end anonymous inner class
);
```

Fig. H.9 Class ElevatorShaft, which represents the ElevatorShaft, which sends events from the Elevator to the ElevatorModel (part 3 of 6).
Fig. H.9 Class ElevatorShaft, which represents the ElevatorShaft, which sends events from the Elevator to the ElevatorModel (part 4 of 6).
```java
204 // get Light on second Floor
205 public Light getSecondFloorLight()
206 {
207     return secondFloorLight;
208 }
209
210 // invoked when Bell rings
211 public void bellRang( BellEvent bellEvent )
212 {
213     bellListener.bellRang( bellEvent );
214 }
215
216 // invoked when Light turns on
217 public void lightTurnedOn( LightEvent lightEvent )
218 {
219     lightListener.lightTurnedOn( lightEvent );
220 }
221
222 // invoked when Light turns off
223 public void lightTurnedOff( LightEvent lightEvent )
224 {
225     lightListener.lightTurnedOff( lightEvent );
226 }
227
228 // invoked when Elevator departs
229 public void elevatorDeparted( ElevatorMoveEvent moveEvent )
230 {
231     Iterator iterator = elevatorMoveListeners.iterator();
232 // iterate Set of ElevatorMoveEvent listeners
233     while ( iterator.hasNext() ) {
234         ElevatorMoveListener listener =
235             ( ElevatorMoveListener ) iterator.next();
236 // get respective ElevatorMoveListener from Set
237         listener.elevatorDeparted( moveEvent );
238     } // end method elevatorDeparted
239
240 // invoked when Elevator arrives
241 public void elevatorArrived( ElevatorMoveEvent moveEvent )
242 {
243     Iterator iterator = elevatorMoveListeners.iterator();
244 // get next DoorListener
245     while ( iterator.hasNext() ) {
246         // obtain iterator from Set
247         ElevatorMoveListener listener =
248             ( ElevatorMoveListener ) iterator.next();
249 // send ElevatorMoveEvent to this listener
250         listener.elevatorArrived( moveEvent );
251     }
252 }
```

Fig. H.9 Class **ElevatorShaft**, which represents the **ElevatorShaft**, which sends events from the **Elevator** to the **ElevatorModel** (part 5 of 6).
The main responsibility of the **ElevatorShaft** is to receive events from other objects then to send these events to the **ElevatorModel** (the **ElevatorModel** then sends the events to the **ElevatorView**, which displays the workings of the model). The **ElevatorShaft** contains references to several different listener objects, such as a **DoorListener**, a **ButtonListener**, a **LightListener**, a **BellListener** and several **ElevatorMoveListeners**. Lines 30–34 declare these listeners—line 34 declares a **Set** to hold multiple **ElevatorMoveListeners**, because the **Buttons**, **Doors**, **Lights** and the **ElevatorModel** are all **ElevatorMoveListeners**. Lines
265–293 provide methods allowing objects to register themselves as listeners for various events.

The constructor (lines 37–166) instantiates listener objects from several inner classes—these listener objects receive events from other objects, then re-send the events to the listener objects defined in line 30–34. For example, lines 43–60 declare a ButtonListener object called floorButtonListener, which contains the logic for when a Button on a Floor has been pressed or reset. Lines 63–70 instantiate firstFloorButton and secondFloorButton, then register floorButtonListener as a ButtonListener for both Button objects. When either Button has been pressed, that Button invokes method buttonPressed (lines 47–53) of the floorButtonListener. When either Button has been reset, that Button invokes the floorButtonListener’s method buttonReset (lines 56–59). Method buttonPressed requests the Elevator by invoking the Elevator’s method requestElevator—method buttonPressed passes a Location reference of the Floor that generated the ButtonEvent. Both methods buttonPressed and buttonReset send the ButtonEvent to the ButtonListener defined in line 31 (which, in this case, is ElevatorModel).

Lines 77–92 declare a DoorListener object called floorDoorListener, which contains the logic for when a Door on a Floor has opened or closed. Lines 95–100 instantiate firstFloorDoor and secondFloorDoor, then register floorDoorListener as a DoorListener for both Door objects. When either Door has opened, that Door calls method doorOpened (lines 80–84) of the floorDoorListener. When either Door has closed, that Door calls method doorClosed (lines 87–91) of the floorDoorListener. Both methods send the DoorEvent to the DoorListener declared in line 30 (which, in this case, is ElevatorModel).

Lines 112–115 instantiate the Elevator and register the ElevatorShaft as an ElevatorMoveListener with the Elevator. When the Elevator has departed, the Elevator invokes method elevatorDeparted (lines 229–243), which informs all objects in elevatorMoveListeners of the departure. When the Elevator has arrived, the Elevator invokes method elevatorArrived (lines 246–262), which informs all objects in elevatorMoveListeners of the arrival.

**H.7 Classes Light and Bell**

Class Light (Fig. H.10) represents the Lights on the Floors in the model. Objects of classes Light help decorate the view by sending events to the ElevatorView via the “bubble up” technique described previously. In our simulation, the ElevatorView turns on and off the Light in the view upon receiving a lightTurnedOn or lightTurnedOff event, respectively.

```java
// Light.java
// Light turns a light on or off
package com.deitel.jhtp4.elevator.model;

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;
```

**Fig. H.10** Class Light represents a Light on the Floor in the model (part 1 of 3).
public class Light implements ElevatorMoveListener {

    // Light state (on/off)
    private boolean lightOn;

    // time before Light turns off automatically (3 seconds)
    public static final int AUTOMATIC_TURNOFF_DELAY = 3000;

    // LightListener listens for when Light should turn on/off
    private LightListener lightListener;

    // location where Light turned on or off
    private Location lightLocation;

    // set LightListener
    public void setLightListener( LightListener listener )
    {
        lightListener = listener;
    }

    // turn on Light
    public void turnOnLight( Location location )
    {
        if ( !lightOn ) {
            lightOn = true;

            // send LightEvent to LightListener
            lightListener.lightTurnedOn(
                new LightEvent( this, location ) );

            lightLocation = location;

            // declare Thread that ensures automatic Light turn off
            Thread thread = new Thread(
                new Runnable() {
                    public void run()
                    {
                        // turn off Light if on for more than 3 seconds
                        try {
                            Thread.sleep( AUTOMATIC_TURNOFF_DELAY );
                            turnOffLight( lightLocation );
                        }
                        catch ( InterruptedException exception ) {
                            exception.printStackTrace();
                        }
                    }
                } // end anonymous inner class
            );
        }
    }
}

Fig. H.10 Class Light represents a Light on the Floor in the model (part 2 of 3).
According to Fig. 15.21, class \texttt{Light} contains attribute \texttt{lightOn} (line 11), which represents the state of the \texttt{Light} (on or off). In addition, Fig. 15.21 specifies that class \texttt{Light} contains methods \texttt{turnOnLight} (lines 29–63) and \texttt{turnOffLight} (lines 66–76). Line 33 of method \texttt{turnOnLight} sets attribute \texttt{lightOn} to \texttt{true}, and lines 36–37 call method \texttt{lightTurnedOn} of the \texttt{lightListener} (line 17). In our model, the \texttt{ElevatorShaft} is the \texttt{lightListener}—the \texttt{ElevatorShaft} receives events from the \texttt{Light} and sends them to the \texttt{ElevatorModel}, which sends them to the \texttt{ElevatorView}. The \texttt{ElevatorShaft} uses method \texttt{setLightListener} (lines 23–26) to register for \texttt{LightEvents}. Method \texttt{turnOffLight} sets attribute \texttt{lightOn} to \texttt{false}, then calls method \texttt{lightTurnedOff} of the \texttt{lightListener}.

We decided in Section 15.12 to make class \texttt{Light} an active class, so that the \texttt{Light} turns itself off after three seconds of being illuminated. Lines 42–59 of method \texttt{turnOn-}
Light instantiate a thread that handles this responsibility. Method run (lines 45–57) puts this thread to sleep for three seconds, then turns off the Light. Line 61 of method turn-OnLight starts the thread.

According to Fig. H.2, class Light implements interface ElevatorMoveListener. Lines 85–89 and lines 92–96 list methods elevatorDeparted and elevatorArrived, respectively. In our model, the Light turns off when the Elevator has departed, and the Light turns on when the Elevator has arrived.

Class Bell (Fig. H.11) represents the Bell in the model and sends a bellRang event to a BellListener when the Bell has rung. This event eventually “bubbles up” to the ElevatorView. The ElevatorView plays an audio clip of a bell ringing upon receiving a bellRang event.

```java
// Bell.java
// Represents Bell in simulation
package com.deitel.jhtp4.elevator.model;

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;

public class Bell implements ElevatorMoveListener {
    private BellListener bellListener;

    private void ringBell( Location location ) {
        if ( bellListener != null )
            bellListener.bellRang( new BellEvent( this, location ) );
    }

    public void setBellListener( BellListener listener ) {
        bellListener = listener;
    }

    public void elevatorDeparted( ElevatorMoveEvent moveEvent ) {}

    public void elevatorArrived( ElevatorMoveEvent moveEvent ) {
        ringBell( moveEvent.getLocation() );
    }
}
```

Fig. H.11 Class Bell represents the Bell in the model.
According to Fig. 15.21, class Bell does not contain attributes, because the Bell does not change state. However, Fig. 15.21 specifies that class Bell contains method ringBell (lines 14–19), which rings the Bell by invoking method bellRang of the BellListener bellListener (line 11). In our simulation, the Elevator is the bellListener—the Elevator receives event from the Bell, then sends the event to the ElevatorShaft, which sends the event to the ElevatorModel, which sends the event to the ElevatorView. The ElevatorView then plays an audio clip of a bell ringing. The Elevator uses method setBellListener (lines 22–24) to register for BellEvents from the Bell.

According to Fig. H.2, class Bell implements interface ElevatorMoveListener. Line 28 and lines 31–34 list methods elevatorDeparted and elevatorArrived, respectively. In our simulation, the Bell rings when the Elevator has arrived.

H.8 Class Elevator

Class Elevator (Fig. H.12) represents the elevator car that travels between the two Floors in the ElevatorShaft while carrying a Person. According to the class diagram of Fig. 15.21, class Elevator contains one object each of classes Button, Door and Bell—lines 37–39 declare objects elevatorButton, elevatorDoor and bell. As discussed in Section 9.23, class Elevator extends superclass Location, because the Elevator is a location that the Person can occupy. Class Elevator implements methods getButton (lines 232–235) and getDoor (lines 238–241) provided by class Location. Method getButton returns the elevatorButton and method getDoor returns the elevatorDoor. According to Fig. 15.21, we also must include two Location objects—one named currentFloor (line 22), which represents the current Floor being serviced, and the other named destinationFloor (line 25), which represents the Floor at which the Elevator will arrive. In addition, Fig. 15.21 specifies that class Elevator requires boolean variable moving (line 19), which describes whether the Elevator is moving or idle, and boolean variable summoned (line 28), which describes whether the Elevator has been summoned. Also, class Elevator uses int constant TRAVEL_TIME (line 44), which indicates the five second travel time between Floors.

```java
// Elevator.java
// Travels between Floors in the ElevatorShaft
package com.deitel.jhtp4.elevator.model;

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

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;
import com.deitel.jhtp4.elevator.ElevatorConstants;

public class Elevator extends Location implements Runnable, BellListener, ElevatorConstants {
```

Fig. H.12  Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 1 of 9).
private boolean elevatorRunning = false;

private boolean moving = false;

private Location currentFloorLocation;

private Location destinationFloorLocation;

private boolean summoned;

private Set elevatorMoveListeners;

private ButtonListener elevatorButtonListener;

private DoorListener elevatorDoorListener;

private BellListener bellListener;

private Door elevatorDoor;

private Button elevatorButton;

private Bell bell;

public static final int ONE_SECOND = 1000;

private static final int TRAVEL_TIME = 5 * ONE_SECOND;

private static final int MAX_TRAVEL_TIME = 20 * 60 * ONE_SECOND;

private Thread thread;

public Elevator( Floor firstFloor, Floor secondFloor )
{
    setLocationName( ELEVATOR_NAME );
    elevatorDoor = new Door();
elevatorButton = new Button();
bell = new Bell();

    bell.setBellListener( this );

Fig. H.12 Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 2 of 9).
// instantiate listener Set
elevatorMoveListeners = new HashSet(1);

// start Elevator on first Floor
currentFloorLocation = firstFloor;
destinationFloorLocation = secondFloor;

// register elevatorButton for ElevatorMoveEvents
addElevatorMoveListener(elevatorButton);

// register elevatorDoor for ElevatorMoveEvents
addElevatorMoveListener(elevatorDoor);

// register bell for ElevatorMoveEvents
addElevatorMoveListener(bell);

// anonymous inner class listens for ButtonEvents from
// elevatorButton
elevatorButton.setButtonListener(new ButtonListener() {
    // invoked when elevatorButton has been pressed
    public void buttonPressed(ButtonEvent buttonEvent) {
        // send ButtonEvent to listener
elevatorButtonListener.buttonPressed(buttonEvent);
        // start moving Elevator to destination Floor
        setMoving(true);
    }

    // invoked when elevatorButton has been reset
    public void buttonReset(ButtonEvent buttonEvent) {
        // send ButtonEvent to listener
elevatorButtonListener.buttonReset(buttonEvent);
    }
} // end anonymous inner class

// anonymous inner class listens for DoorEvents from
// elevatorDoor
elevatorDoor.addDoorListener(new DoorListener() {
    // invoked when elevatorDoor has opened
    public void doorOpened(DoorEvent doorEvent) {
        // get Location associated with DoorEvent
        Location location = doorEvent.getLocation();
    }
});

Fig. H.12 Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 3 of 9).
String locationName = location.getLocationName();

// open Door on Floor Location
if ( !locationName.equals( ELEVATOR_NAME ) )
    location.getDoor().openDoor( location );

// send DoorEvent to listener
elevatorDoorListener.doorOpened( new DoorEvent(
    doorEvent.getSource(), Elevator.this ));

// invoked when elevatorDoor has closed
public void doorClosed( DoorEvent doorEvent )
{
    // get Location associated with DoorEvent
    Location location = doorEvent.getLocation();
    String locationName = location.getLocationName();

    // close Door on Floor Location
    if ( !locationName.equals( ELEVATOR_NAME ) )
        location.getDoor().closeDoor( location );

    // send DoorEvent to listener
elevatorDoorListener.doorClosed( new DoorEvent(
        doorEvent.getSource(), Elevator.this ));
}

// swaps current Floor Location with opposite Floor Location
private void changeFloors()
{
    Location location = currentFloorLocation;
    currentFloorLocation = destinationFloorLocation;
    destinationFloorLocation = location;
}

// start Elevator thread
public void start()
{
    if ( thread == null )
        thread = new Thread( this );

    elevatorRunning = true;
    thread.start();
}
public void stopElevator()
{
    elevatorRunning = false;
}

public void run()
{
    while ( isElevatorRunning() ) {
        // remain idle until awoken
        while ( !isMoving() )
            pauseThread( 10 );
        // close elevatorDoor
        getDoor().closeDoor( currentFloorLocation );
        // closing Door takes one second
        pauseThread( ONE_SECOND );
        // issue elevatorDeparted Event
        sendDepartureEvent( currentFloorLocation );
        // Elevator needs 5 seconds to travel
        pauseThread( TRAVEL_TIME );
        // stop Elevator
        setMoving( false );
        // swap Floor Locations
        changeFloors();
        // issue elevatorArrived Event
        sendArrivalEvent( currentFloorLocation );
    } // end while loop
}

public synchronized void ride()
{
    try {
        Thread.sleep( MAX_TRAVEL_TIME );
    } 
    catch ( InterruptedException exception ) {
        // method doorOpened in Person interrupts method sleep;
        // Person has finished riding Elevator
    }
}

Fig. H.12 Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 5 of 9).
// pause concurrent thread for number of milliseconds
private void pauseThread( int milliseconds )
{
    try {
        Thread.sleep( milliseconds );
    }
    // handle if interrupted while sleeping
    catch ( InterruptedException exception ) {
        exception.printStackTrace();
    }
} // end method pauseThread

// return Button on Elevator
public Button getButton()
{
    return elevatorButton;
}

// return Door on Elevator
public Door getDoor()
{
    return elevatorDoor;
}

// set if Elevator should move
public void setMoving( boolean elevatorMoving )
{
    moving = elevatorMoving;
}

// is Elevator moving?
public boolean isMoving()
{
    return moving;
}

// is Elevator thread running?
private boolean isElevatorRunning()
{
    return elevatorRunning;
}

// register ElevatorMoveListener for ElevatorMoveEvents
public void addElevatorMoveListener( ElevatorMoveListener listener )
{
    elevatorMoveListeners.add( listener );
}

Fig. H.12 Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 6 of 9).
268 // register ButtonListener for ButtonEvents
269 public void setButtonListener( ButtonListener listener )
270 {
271   elevatorButtonListener = listener;
272 }
273
274 // register DoorListener for DoorEvents
275 public void setDoorListener( DoorListener listener )
276 {
277   elevatorDoorListener = listener;
278 }
279
280 // register BellListener for BellEvents
281 public void setBellListener( BellListener listener )
282 {
283   bellListener = listener;
284 }
285
286 // notify all ElevatorMoveListeners of arrival
287 private void sendArrivalEvent( Location location )
288 {
289   // obtain iterator from Set
290   Iterator iterator = elevatorMoveListeners.iterator();
291
292   // get next DoorListener
293   while ( iterator.hasNext() ) {
294
295     // get next ElevatorMoveListener from Set
296     ElevatorMoveListener listener =
297       ( ElevatorMoveListener ) iterator.next();
298
299     // send event to listener
300     listener.elevatorArrived( new 
301       ElevatorMoveEvent( this, location ) );
302
303   } // end while loop
304
305   // service queued request, if one exists
306   if ( summoned ) {
307     pauseThread( Door.AUTOMATIC_CLOSE_DELAY );
308     setMoving( true ); // start moving Elevator
309   }
310
311   summoned = false; // request has been serviced
312
313 } // end method sendArrivalEvent
314
315 // notify all ElevatorMoveListeners of departure
316 private void sendDepartureEvent( Location location )
317 {
318   // obtain iterator from Set
319   Iterator iterator = elevatorMoveListeners.iterator();
320

Fig. H.12 Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 7 of 9).
320 // get next DoorListener
321 while ( iterator.hasNext() ) {
322
323 // get next ElevatorMoveListener from Set
324 ElevatorMoveListener listener =
325 ( ElevatorMoveListener ) iterator.next();
326
327 // send ElevatorMoveEvent to this listener
328 listener.elevatorDeparted( new ElevatorMoveEvent(
329 this, currentFloorLocation ) );
330
331 } // end while loop
332 } // end method sendDepartureEvent
333
334 // request Elevator
335 public void requestElevator( Location location )
336 {
337
338 // if Elevator is idle
339 if ( !isMoving() ) {
340
341 // if Elevator is on same Floor of request
342 if ( location == currentFloorLocation )
343
344 // Elevator has already arrived; send arrival event
345 sendArrivalEvent( currentFloorLocation );
346
347 // if Elevator is on opposite Floor of request
348 else {
349
350 if ( getDoor().isDoorOpen() )
351 pauseThread( Door.AUTOMATIC_CLOSE_DELAY );
352 setMoving( true ); // move to other Floor
353 }
354 }
355 else // if Elevator is moving
356
357 // if Elevator departed from same Floor as request
358 if ( location == currentFloorLocation )
359 summoned = true;
360
361 // if Elevator is traveling to Floor of request,
362 // continue traveling
363
364 } // end method requestElevator
365
366 // invoked when bell has rung
367 public void bellRang( BellEvent bellEvent )
368 {
369 // send event to bellListener
370 if ( bellListener != null )
371 bellListener.bellRang( bellEvent );
372
Fig. H.12 Class Elevator represents the Elevator traveling between two Floors, operating asynchronously with other objects (part 8 of 9).
According to Fig. H.2, class `Elevator` implements interfaces `ButtonListener`, `DoorListener` and `BellListener` and therefore can listen for `ButtonEvents`, `DoorEvents` and `BellEvents`. Class `Elevator` must send these events to a listener (in this case, the `ElevatorShaft`), so that these events can “bubble up” to the `ElevatorView`. Class `Elevator` contains a `ButtonListener` called `elevatorButtonListener` (line 32), a `DoorListener` called `elevatorDoorListener` (line 33) and a `BellListener` called `bellListener` (line 34). Lines 269–284 list methods `setButtonListener`, `setDoorListener` and `setBellListener` that allow an object—such as `ElevatorShaft`—to register as a listener for these events.

Class `Elevator` contains an anonymous `ButtonListener` (lines 84–106) that registers for `ButtonEvents` from the `elevatorButton`. When a `Person` has pressed the `elevatorButton`, the `ButtonListener` calls method `buttonPressed` (lines 88–96) of the `ButtonListener`. Lines 91–92 of this method call method `buttonPressed` of the `elevatorButtonListener`, and line 95 informs the `Elevator` to move using method `setMoving`. When the `Button` has been reset, the `ButtonListener` calls method `buttonReset` (lines 99–104) of the `ButtonListener`. Lines 102–103 of this method calls method `buttonReset` of the `elevatorButtonListener`.

Class `Elevator` contains an anonymous `DoorListener` (lines 110–145) that registers for `DoorEvents` from the `elevatorDoor`. When the `elevatorDoor` has opened, the `DoorListener` calls method `doorOpened` (lines 114–127) of this `DoorListener`. Lines 121–122 open the `Door` on the `Floor` that generated the event, and lines 125–126 call method `doorOpened` of the `elevatorDoorListener`. Method `doorOpened` guarantees that the `Door` on the `Floor` opens before the passenger exits the `Elevator`. When the `elevatorDoor` has closed, the `DoorListener` calls method `doorClosed` (lines 130–143) of the `DoorListener`. Lines 137–138 close the `Door` on the `Floor` that generated the event, and lines 141–142 call method `doorClosed` of the `elevatorDoorListener`.

Class `Elevator` acts as a thread because it implements interface `Runnable`. Method `run` (lines 173–204) handles the travel between `Floors`. The method begins with the `Elevator` remaining idle in a `while` loop (lines 178–179). The loop exits when method `buttonPressed` in the anonymous `ButtonListener` calls method `setMoving`.

When the `Elevator` exits the loop, the `Elevator` closes the `elevatorDoor` (line 182) then calls private method `sendDepartureEvent` (lines 316–333), to inform all listeners—the `elevatorButton`, the `elevatorDoor`, the `bell` and the `ElevatorShaft`—of the `Elevator`’s departure. Class `Elevator` contains Set `elevatorMoveListeners` (line 31), which stores all registered `ElevatorMoveListeners`. Objects wishing to receive `ElevatorMoveEvents` from the `Elevator` must call method `addElevatorMoveListener` (lines 262–266), which appends that object to `elevatorMoveListeners`. Method `sendDepartureEvent` invokes method `elevatorDeparted` of each listener object in Set `elevatorMoveListeners`. 
Line 191 of method `run` allows the `Elevator` to travel to the `Floor` by calling method `pauseThread` (lines 219–229)—this simulates travel by invoking method `sleep` of class `Thread`. The `Elevator` stops moving when its thread awakens after five seconds. Line 197 calls private method `changeFloors` (lines 149–154), which swaps `currentFloorLocation` and `destinationFloorLocation`. Line 200 calls private method `sendArrivalEvent` (lines 287–313), which invokes method `elevatorArrived` of all listeners in `Set elevatorMoveListeners`. Lines 306–309 of method `sendArrivalEvent` service any queued request (e.g., a `Person` pressed a `Button` on the `Floor` from which the `Elevator` has departed). If a queued request exists, line 308 invokes method `setMoving` to move the `Elevator` to the opposite `Floor`.

Method `requestElevator` (lines 336–364) requests the `Elevator` and generates a queued request. In our model, the `ButtonListener` defined in the inner class of the `ElevatorShaft` calls this method when a `Button` on either `Floor` has been pressed. The activity diagram of Fig. 5.30 specifies the logic for method `requestElevator`. If the `Elevator` is idle and on the same `Floor` as the `Floor` of the request, line 345 calls method `sendArrivalEvent`, because the `Elevator` has already arrived. If the `Elevator` is idle but on the opposite `Floor` from the `Floor` of the request, line 352 moves the `Elevator` to the opposite `Floor`. If the `Elevator` is traveling to the `Floor` that generated the request, the `Elevator` should continue traveling to that `Floor`. If the `Elevator` is traveling away from the `Floor` that generated the request, the `Elevator` must remember to return to that `Floor` (lines 358–359).

Lastly, as mentioned in Section 15.12, class `Elevator` contains synchronized method `ride` (lines 207–216). The `Person` calls this method to guarantee exclusivity with the `Elevator`. Method `ride` ensures that two `Person` objects cannot occupy the `Elevator` at the same time. When a `Person` object invokes method `ride`, that `Person` object obtains a monitor on the `Elevator` object. Other objects may not access the `Elevator` until that `Person` releases the monitor by exiting method `ride`.

### H.9 Class `Person`

Class `Person` (Fig. H.13) represents a `Person` that walks across the `Floors` and rides the `Elevator` in our simulation. According to the class diagram of Fig. 15.21, class `Person` contains one object of class `Location` (line 20) that represents the `Person`’s current location in the model (either on a `Floor` or in the `Elevator`). In addition, Fig. 15.21 specifies that `Person` requires `int` attribute `ID` (line 14) as a unique identifier and `boolean` attribute `moving` (line 17), which indicates whether `Person` is walking across the `Floor` or waiting for a `Door` to open.

```java
1  // Person.java
2  // Person riding the elevator
3  package com.deitel.jhtp4.elevator.model;
4
5  // Java core packages
6  import java.util.*;
```

Fig. H.13 Class `Person` represents the `Person` that rides the `Elevator`. The `Person` operates asynchronously with other objects (part 1 of 7).
// Deitel packages
import com.deitel.jhtp4.elevator.event.*;

public class Person extends Thread implements DoorListener {

    // identification number
    private int ID = -1;

    // represents whether Person is moving or waiting
    private boolean moving;

    // reference to Location (either on Floor or in Elevator)
    private Location location;

    // listener object for PersonMoveEvents
    private PersonMoveListener personMoveListener;

    // time in milliseconds to walk to Button on Floor
    private static final int TIME_TO_WALK = 3000;

    // maximum time Person will wait for Elevator (10 minutes)
    private static final int TIME_WAITING = 10 * 60 * 1000;

    // types of messages Person may send
    public static final int PERSON_CREATED = 1;
    public static final int PERSON_ARRIVED = 2;
    public static final int PERSON_ENTERING_ELEVATOR = 3;
    public static final int PERSON_PRESSING_BUTTON = 4;
    public static final int PERSON_EXITING_ELEVATOR = 5;
    public static final int PERSON_EXITED = 6;

    // Person constructor set initial location
    public Person( int identifier, Location initialLocation )
    {
        super();

        ID = identifier; // assign unique identifier
        location = initialLocation; // set Floor Location
        moving = true; // start moving toward Button on Floor
    }

    // set listener for PersonMoveEvents
    public void setPersonMoveListener( PersonMoveListener listener )
    {
        personMoveListener = listener;
    }

    // invoked when Door has opened
    public void doorOpened( DoorEvent doorEvent )
    {
        // handle PersonMoveEvent
        if (doorEvent instanceof PersonMoveEvent)
        {
            PersonMoveEvent event = (PersonMoveEvent) doorEvent;
            moving = event.isMoving();
        }
    }
}
// set Person on Floor where Door opened
setLocation( doorEvent.getLocation() );

// interrupt Person’s sleep method in run method and
// Elevator’s ride method
interrupt();
}

// invoked when Door has closed
public void doorClosed( DoorEvent doorEvent ) {}  

// set Person Location
private void setLocation( Location newLocation )
{
    location = newLocation;
}

// get current Location
private Location getLocation()
{
    return location;
}

// get identifier
public intgetID()
{
    return ID;
}

// set if Person should move
public void setMoving( boolean personMoving )
{
    moving = personMoving;
}

// get if Person should move
public boolean isMoving()
{
    return moving;
}

// Person either rides or waits for Elevator
public void run()
{
    sendPersonMoveEvent( PERSON CREATED );
    
    // walk to Elevator
    pauseThread( TIME TO WALK );
    setMoving( false );
    
    // Person arrived at Floor Button
    sendPersonMoveEvent( PERSON ARRIVED );

Fig. H.13 Class Person represents the Person that rides the Elevator. The Person operates asynchronously with other objects (part 3 of 7).
// get current Door on Floor
Door currentFloorDoor = location.getDoor();

// determine if Door on Floor is open
try {
    boolean doorOpen = currentFloorDoor.isDoorOpen();

    // if Door on Floor is closed
    if ( !doorOpen ) {
        // press Floor Button
        sendPersonMoveEvent( PERSON_PRESSING_BUTTON );
        pauseThread( 1000 );

        // register for Floor Door's doorOpen event
        currentFloorDoor.addDoorListener( this );

        // press Floor's Button to request Elevator
        Button floorButton = getLocation().getButton();
        floorButton.pressButton( getLocation() );

        // wait for Floor's Door to open
        sleep( TIME_WAITING );

        // unregister with Floor's Door if too long
        currentFloorDoor.removeDoorListener( this );
    }

    // if Door on Floor is open, ride Elevator
    else
        enterAndRideElevator();
}

// handle exception when interrupted from waiting
catch ( InterruptedException interruptedException ) {
    // Person unregisters for Floor's Door doorOpen event
    currentFloorDoor.removeDoorListener( this );

    // enter and ride Elevator when Door on Floor opens,
    pauseThread( 1000 );
    enterAndRideElevator();
}

// waiting for Elevator's Door to open takes a second
pauseThread( 1000 );

// begin walking away from Elevator
setMoving( true );

Fig. H.13 Class Person represents the Person that rides the Elevator. The Person operates asynchronously with other objects (part 4 of 7).
// Person exits Elevator
sendPersonMoveEvent(PERSON_EXITING_ELEVATOR);

// walking from elevator takes five seconds
pauseThread(2 * TIME_TO_WALK);

// Person exits simulation
sendPersonMoveEvent(PERSON_EXITED);

} // end method run

// Person enters Elevator
private void enterAndRideElevator()
{
    // Person enters Elevator
sendPersonMoveEvent(PERSON_ENTERING_ELEVATOR);

    // set Person Location to Elevator
Floor floorLocation = (Floor)getLocation();
setLocation(floorLocation.getElevatorShaft().getElevator());

    // Person takes one second to enter Elevator
pauseThread(1000);

    // register for Elevator's Door's doorOpen event
Door elevatorDoor = getLocation().getDoor();
elevatorDoor.addDoorListener(this);

    // pressing Elevator Button takes one second
sendPersonMoveEvent(PERSON_PRESSING_BUTTON);
pauseThread(1000);

    // get Elevator's Button
Button elevatorButton = getLocation().getButton();

    // press Elevator's Button
elevatorButton.pressButton(location);

    // Door closing takes one second
pauseThread(1000);

    // ride in Elevator
Elevator elevator = (Elevator)getLocation();
elevator.ride();

    // Person finished riding Elevator
    // unregister for Elevator's Door's doorOpen event
elevatorDoor.removeDoorListener(this);
}

} // end method enterAndRideElevator

Fig. H.13 Class Person represents the Person that rides the Elevator. The Person operates asynchronously with other objects (part 5 of 7).
// pause thread for desired number of milliseconds
private void pauseThread( int milliseconds )
{
    try {
        sleep( milliseconds );
    }

    // handle exception if interrupted when paused
    catch ( InterruptedException interruptedException ) {
        interruptedException.printStackTrace();
    }
} // end method pauseThread

// send PersonMoveEvent to listener, depending on event type
private void sendPersonMoveEvent( int eventType )
{
    // create new event
    PersonMoveEvent event =
        new PersonMoveEvent( this, getLocation(), getID() );

    // send Event to this listener, depending on eventType
    switch ( eventType ) {

        // Person has been created
        case PERSON_CREATED:
            personMoveListener.personCreated( event );
            break;

        // Person arrived at Elevator
        case PERSON_ARRIVED:
            personMoveListener.personArrived( event );
            break;

        // Person entered Elevator
        case PERSON_ENTERING_ELEVATOR:
            personMoveListener.personEntered( event );
            break;

        // Person pressed Button object
        case PERSON_PRESSING_BUTTON:
            personMoveListener.personPressedButton( event );
            break;

        // Person exited Elevator
        case PERSON_EXITING_ELEVATOR:
            personMoveListener.personDeparted( event );
            break;

        // Person exited simulation
        case PERSON_EXITED:
            personMoveListener.personExited( event );
    }
} // end method sendPersonMoveEvent

Fig. H.13 Class Person represents the Person that rides the Elevator. The Person operates asynchronously with other objects (part 6 of 7).
Class **Person** is subclass of class **Thread**. The **Person** performs all actions, such as walking across **Floors** and riding the **Elevator**, in method **run** (lines 101–172). Method **run** represents the lifetime of a **Person** described in the sequence diagram of Fig. 15.20. Class **Person** contains a **PersonMoveListener** object (line 23) to which the **Person** sends **PersonMoveEvent**s. In our simulation, the **ElevatorModel** uses method **setPersonMoveListener** (lines 50–54) to register itself as the **PersonMoveListener**. The **ElevatorModel**, upon receiving a **PersonMoveEvent**, sends the event to the **ElevatorView**—therefore, the **ElevatorView** “knows” when a **Person** has performed certain actions discussed momentarily.

There are several types of actions a **Person** performs in its lifetime, so there exists several types of **PersonMoveEvent**s that a **Person** may send to the **personMoveListener**. Lines 32–37 define a series of constants in which each constant represents a unique type of **PersonMoveEvent**. The **Person** sends events to **personMoveListener** when

- the **Person** has been created
- the **Person** arrives at the **Elevator**
- the **Person** enters the **Elevator**
- the **Person** presses a **Button** (either in the **Elevator** or on a **Floor**)
- the **Person** exits the **Elevator**
- the **Person** exits the simulation

When the **Person** decides to send an event to its **PersonMoveListener**, the **Person** calls private method **sendPersonMoveEvent** and passes the desired constant as a parameter. This method sends the event associated with the constant. For example, line 124 calls

```java
sendPersonMoveEvent( PERSON_PRESSED_BUTTON );
```

when the **Person** presses a **Button** on a **Floor**. In method **run**, the **Person** walks to the **Elevator**, then sends a **personArrived** event upon arrival at the **Elevator**. We use the activity diagram of Fig. 5.29 to determine the **Person**’s next action. If the **Door** on the **Floor** is closed (line 121), the **Person** must wait for that **Door** to open. Specifically, line 128 registers the **Person** as a **DoorListener** for that **Door**, and lines 131–132 allow the **Person** to press the **Button** on that **Floor**. The **Person** waits for the **Door** to open by calling method **sleep** (line 135) of the **Person**’s superclass **Thread**. When the **Door** opens, that **Door** informs **Person** in **doorOpened**.
(lines 57–65). Line 64 of method `doorOpened` interrupts the `Person`’s thread, which terminates method `sleep` of line 135. Method `interrupt` throws an `InterruptedException` received by a `catch` block (lines 147–155). Line 150 of this `catch` block unregisters the `Person` with the `Door` on the `Floor`, and line 154 calls private method `enterAndRideElevator` (lines 175–214)—however, according to Fig. 5.29, if the `Door` on the `Floor` is already open when the `Person` arrives, the `Person` calls method `enterAndRideElevator` immediately.

Line 178 of method `enterAndRideElevator` sends a `personEntered` event to `personMoveListener`, indicating that the `Person` is entering the `Elevator`. When the `Person` enters the `Elevator`, the `Person`’s `Location` references the `Elevator` (lines 181–183). When the `Person` has entered the `Elevator`, lines 189–190 register the `Person` as a `DoorListener` with the `Door` in the `Elevator`. Lines 197–200 allow the `Person` to press the `Button` in the `Elevator` and to send a `personPressedButton` event to the `personMoveListener`. Lines 206–207 invoke the `synchronized` method `ride` in the `Elevator`, ensuring that other `Persons` cannot occupy the `Elevator`. When the `Elevator` arrives, it sends an `elevatorArrived` event to the `Door` in the `Elevator`, which opens that `Door` and invokes method `doorOpened` of the `Person`. Method `doorOpened`, as previously mentioned, interrupts the `Person`’s thread—in this case, method `interrupt` terminates the sleep in method `ride` and allows the `Person` to exit the `Elevator` (allowing a `Person` waiting for the `Elevator` to enter). Method `enterAndRideElevator` returns, and lines 161–170 of method `run` make the `Person` exit the `Elevator` and exit the simulation shortly thereafter.

### H.10 Component Diagrams Revisited

In Section 13.17, we introduced the component diagram for the elevator simulation. In our simulation, the each class in the model imports package `event`—we showed the components of package `event` in Fig. G.15. Figure H.14 presents the component diagram for package `model`. Each component in package `model` maps to a class from the class diagram of Fig. 15.21—package `model` aggregates package `event`.

### H.11 Conclusion

This concludes discussion case study model. We hope you have enjoyed the design process of our elevator simulation using the UML, along with the presentation of object-oriented fundamentals and Java-specific topics, such as event handling and multithreading. Using the concepts discussed in this case study, you should now be able to tackle even larger systems. We encourage you to read Appendix I, which implements the `ElevatorView`, which transforms the `ElevatorModel` that we have designed into a vibrant and interactive program abundant with graphics, animation and sound.
Fig. H.14  Component diagram for package model.
I.1 Introduction

This appendix contains the implementation for class ElevatorView (Fig. I.1). Familiarity with the “Thinking About Objects” sections from all chapters (Chapter 22, in particular) necessitates the understanding of material presented in this appendix. Class ElevatorView is the largest class in the simulation. To facilitate discussion, we have divided the discussion of the ElevatorView into five topics—Class Objects, Class Constants, Class Constructor, Event Handling and Component Diagrams Revisited.

```java
// ElevatorView.java
// View for ElevatorSimulation
package com.deitel.jhtp4.elevator.view;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.applet.*;

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

// Deitel packages
import com.deitel.jhtp4.elevator.event.*;
import com.deitel.jhtp4.elevator.ElevatorConstants;

public class ElevatorView extends JPanel
    implements ActionListener, ElevatorModelListener,
              ElevatorConstants {

Fig. I.1 ElevatorView displays the elevator simulation model (part 1 of 18).
// ElevatorView dimensions
private static final int VIEW_WIDTH = 800;
private static final int VIEW_HEIGHT = 435;

// offset for positioning Panels in ElevatorView
private static final int OFFSET = 10;

// Elevator repaints components every 50 ms
private static final int ANIMATION_DELAY = 50;

// horizontal distance constants
private static final int PERSON_TO_BUTTON_DISTANCE = 400;
private static final int BUTTON_TO_ELEVATOR_DISTANCE = 50;
private static final int PERSON_TO_ELEVATOR_DISTANCE =
    PERSON_TO_BUTTON_DISTANCE + BUTTON_TO_ELEVATOR_DISTANCE;

// times walking to Floor's Button and Elevator
private static final int TIME_TO_BUTTON = 3000; // 3 seconds
private static final int TIME_TO_ELEVATOR = 1000; // 1 second

// time traveling in Elevator (5 seconds)
private static final int ELEVATOR_TRAVEL_TIME = 5000;

// Door images for animation
private static final String doorFrames[] = {
    "images/door1.png", "images/door2.png", "images/door3.png",
    "images/door4.png", "images/door5.png"};

// Person images for animation
private static final String personFrames[] = {
    "images/bug1.png", "images/bug2.png", "images/bug3.png",
    "images/bug4.png", "images/bug5.png", "images/bug6.png",
    "images/bug7.png", "images/bug8.png"};

// Light images for animation
private static final String lightFrames[] = {
    "images/lightOff.png", "images/lightOn.png"};

// Floor Light images for animation
private static final String firstFloorLightFrames[] = {
    "images/firstFloorLightOff.png",
    "images/firstFloorLightOn.png"};

private static final String secondFloorLightFrames[] = {
    "images/secondFloorLightOff.png",
    "images/secondFloorLightOn.png",};

// Floor Button images for animation
private static final String floorButtonFrames[] = {
    "images/floorButtonUnpressed.png",
    "images/floorButtonPressed.png",
    "images/floorButtonLit.png"};

Fig. I.1 ElevatorView displays the elevator simulation model (part 2 of 18).
// Elevator Button images for animation
private static final String elevatorButtonFrames[] = {
    "images/elevatorButtonUnpressed.png",
    "images/elevatorButtonPressed.png",
    "images/elevatorButtonLit.png" }; 

// Bell images for animation
private static final String bellFrames[] = {
    "images/bell1.png", "images/bell2.png",
    "images/bell3.png" };

private static final String floorImage =
    "images/floor.png"; 
private static final String ceilingImage =
    "images/ceiling.png";
private static final String elevatorImage =
    "images/elevator.png";
private static final String wallImage =
    "images/wall.jpg";
private static final String elevatorShaftImage =
    "images/elevatorShaft.png";

// audio files
private static final String bellSound = "bell.wav";
private static final String doorOpenSound = "doorOpen.wav";
private static final String doorCloseSound = "doorClose.wav";
private static final String elevatorSound = "elevator.au";
private static final String buttonSound = "button.wav";
private static final String walkingSound = "walk.wav";
private static final String midiFile = "sounds/liszt.mid";

// ImagePanels for Floors, ElevatorShaft, wall and ceiling
private ImagePanel firstFloorPanel;
private ImagePanel secondFloorPanel;
private ImagePanel elevatorShaftPanel;
private ImagePanel wallPanel;
private ImagePanel ceilingPanel;

// MovingPanels for Elevator
private MovingPanel elevatorPanel;

// AnimatedPanels for Buttons, Bell, Lights and Door
private AnimatedPanel firstFloorButtonPanel;
private AnimatedPanel secondFloorButtonPanel;
private AnimatedPanel elevatorButtonPanel;
private AnimatedPanel bellPanel;
private AnimatedPanel elevatorLightPanel;
private AnimatedPanel firstFloorLightPanel;
private AnimatedPanel secondFloorLightPanel;
private AnimatedPanel doorPanel;

Fig. I.1 ElevatorView displays the elevator simulation model (part 3 of 18).
// List containing AnimatedPanels for all Person objects
private java.util.List personAnimatedPanels;

// AudioClips for sound effects
private AudioClip bellClip;
private AudioClip doorOpenClip;
private AudioClip doorCloseClip;
private AudioClip elevatorClip;
private AudioClip buttonClip;
private AudioClip walkClip;

// ElevatorMusic to play in Elevator
private ElevatorMusic elevatorMusic;

// Timer for animation controller;
private javax.swing.Timer animationTimer;

// distance from top of screen to display Floors
private int firstFloorPosition;
private int secondFloorPosition;

// Elevator's velocity
private double elevatorVelocity;

// ElevatorView constructor
public ElevatorView()
{
    // specify null Layout
    super( null );

    instantiatePanels();
    placePanelsOnView();
    initializeAudio();

    // calculate distance Elevator travels
    double floorDistance =
        firstFloorPosition - secondFloorPosition;

    // calculate time needed for travel
    double time = ELEVATOR_TRAVEL_TIME / ANIMATION_DELAY;

    // determine Elevator velocity (rate = distance / time)
    elevatorVelocity = ( floorDistance + OFFSET ) / time;

    // start animation Thread
    startAnimation();

    } // end ElevatorView constructor

// instantiate all Panels (Floors, Elevator, etc.)
private void instantiatePanels()
{

Fig. I.1 ElevatorView displays the elevator simulation model (part 4 of 18).
// instantiate ImagePanels representing Floors
firstFloorPanel = new ImagePanel( 0, floorImage );
secondFloorPanel = new ImagePanel( 0, floorImage );

// calculate first and second Floor positions
firstFloorPosition = VIEW_HEIGHT - firstFloorPanel.getHeight();
secondFloorPosition =
    ( int )( firstFloorPosition / 2 ) - OFFSET;

firstFloorPanel.setPosition( 0, firstFloorPosition );
secondFloorPanel.setPosition( 0, secondFloorPosition );

wallPanel = new ImagePanel( 0, wallImage );

// create and position ImagePanel for ElevatorShaft
elevatorShaftPanel =
    new ImagePanel( 0, elevatorShaftImage );

double xPosition = PERSON_TO_ELEVATOR_DISTANCE + OFFSET;
double yPosition =
    firstFloorPosition - elevatorShaftPanel.getHeight();
elevatorShaftPanel.setPosition( xPosition, yPosition );

// create and position ImagePanel for ceiling
ceilingPanel = new ImagePanel( 0, ceilingImage );
yPosition = elevatorShaftPanel.getPosition().getY() -
    ceilingPanel.getHeight();

ceilingPanel.setPosition( xPosition, yPosition );

// create and position MovingPanel for Elevator
elevatorPanel = new MovingPanel( 0, elevatorImage );
yPosition = firstFloorPosition - elevatorPanel.getHeight();
elevatorPanel.setPosition( xPosition, yPosition );

// create and position first Floor Button
firstFloorButtonPanel =
    new AnimatedPanel( 0, floorButtonFrames );
xPosition = PERSON_TO_BUTTON_DISTANCE + 2 * OFFSET;
yPosition = firstFloorPosition - 5 * OFFSET;
firstFloorButtonPanel.setPosition( xPosition, yPosition );

int floorButtonPressedFrameOrder[] = { 0, 1, 2 };
firstFloorButtonPanel.addFrameSequence( floorButtonPressedFrameOrder );

Fig. I.1 ElevatorView displays the elevator simulation model (part 5 of 18).
// create and position second Floor Button
secondFloorButtonPanel =
    new AnimatedPanel( 1, floorButtonFrames );
xPosition = PERSON_TO_BUTTON_DISTANCE + 2 * OFFSET;
yPosition = secondFloorPosition - 5 * OFFSET;
secondFloorButtonPanel.setPosition( xPosition, yPosition );
secondFloorButtonPanel.addFrameSequence(
    floorButtonPressedFrameOrder );

// create and position Floor Lights
firstFloorLightPanel =
    new AnimatedPanel( 0, firstFloorLightFrames );
xPosition = elevatorPanel.getLocation().x - 4 * OFFSET;
yPosition =
    firstFloorButtonPanel.getLocation().y - 10 * OFFSET;
firstFloorLightPanel.setPosition( xPosition, yPosition );

secondFloorLightPanel =
    new AnimatedPanel( 1, secondFloorLightFrames );
yPosition =
    secondFloorButtonPanel.getLocation().y - 10 * OFFSET;
secondFloorLightPanel.setPosition( xPosition, yPosition );

// create and position Door AnimatedPanels
doorPanel = new AnimatedPanel( 0, doorFrames );
doorOpenedFrameOrder[] = { 0, 1, 2, 3, 4 };
doorClosedFrameOrder[] = { 4, 3, 2, 1, 0 };
doorPanel.addFrameSequence( doorOpenedFrameOrder );
doorPanel.addFrameSequence( doorClosedFrameOrder );

// determine where Door is located relative to Elevator
yPosition =
    elevatorPanel.getHeight() - doorPanel.getHeight();
doorPanel.setPosition( 0, yPosition );

// create and position Light AnimatedPanel
elevatorLightPanel = new AnimatedPanel( 0, lightFrames );
elevatorLightPanel.setPosition( OFFSET, 5 * OFFSET );

// create and position Bell AnimatedPanel
bellPanel = new AnimatedPanel( 0, bellFrames );
yPosition = elevatorLightPanel.getPosition().getY() +
elevatorLightPanel.getHeight() + OFFSET;
bellPanel.setPosition( OFFSET, yPosition );
bellRingAnimation[] = { 0, 1, 0, 2 };
bellPanel.addFrameSequence( bellRingAnimation );

Fig. I.1 Elevator View displays the elevator simulation model (part 6 of 18).
// create and position Elevator's Button AnimatedPanel
elevatorButtonPanel =
    new AnimatedPanel( 0, elevatorButtonFrames );
yPosition = elevatorPanel.getHeight() - 6 * OFFSET;
elevatorButtonPanel.setPosition( 10 * OFFSET, yPosition );

int buttonPressedFrameOrder[] = { 0, 1, 2 };
elevatorButtonPanel.addFrameSequence( buttonPressedFrameOrder );

// create List to store Person AnimatedPanels
personAnimatedPanels = new ArrayList();

} // end method instantiatePanels

// place all Panels on ElevatorView
private void placePanelsOnView()
{
    // add Panels to ElevatorView
    add( firstFloorPanel );
    add( secondFloorPanel );
    add( ceilingPanel );
    add( elevatorPanel );
    add( firstFloorButtonPanel );
    add( secondFloorButtonPanel );
    add( firstFloorLightPanel );
    add( secondFloorLightPanel );
    add( elevatorShaftPanel );
    add( wallPanel );

    // add Panels to Elevator's MovingPanel
elevatorPanel.add( doorPanel );
elevatorPanel.add( elevatorLightPanel );
elevatorPanel.add( bellPanel );
elevatorPanel.add( elevatorButtonPanel );

} // end method placePanelsOnView

// get sound effects and elevatorMusic
private void initializeAudio()
{
    // create AudioClip sound effects from audio files
    SoundEffects sounds = new SoundEffects();
sounds.setPathPrefix( "sounds/" );

    bellClip = sounds.getAudioClip( bellSound );
doorOpenClip = sounds.getAudioClip( doorOpenSound );
doorCloseClip = sounds.getAudioClip( doorCloseSound );
elevatorClip = sounds.getAudioClip( elevatorSound );
buttonClip = sounds.getAudioClip( buttonSound );
walkClip = sounds.getAudioClip( walkingSound );

Fig. I.1 ElevatorView displays the elevator simulation model (part 7 of 18).
Appendix I

Elevator View (on CD)

// create MIDI player using Java Media Framework
ElevatorMusic elevatorMusic = new ElevatorMusic(midiFile);
elevatorMusic.open();

} // end method initializeAudio

// starts animation by repeatedly drawing images to screen
public void startAnimation()
{
    if (animationTimer == null) {
        animationTimer =
            new javax.swing.Timer(ANIMATION_DELAY, this);
        animationTimer.start();
    } else
        if (!animationTimer.isRunning())
            animationTimer.restart();
}

// stop animation
public void stopAnimation()
{
    animationTimer.stop();
}

// update AnimatedPanels animation in response to Timer
public void actionPerformed(ActionEvent actionEvent)
{
    elevatorPanel.animate();
    firstFloorButtonPanel.animate();
    secondFloorButtonPanel.animate();
    Iterator iterator = getPersonAnimatedPanelsIterator();
    while (iterator.hasNext()) {
        AnimatedPanel personPanel =
            (AnimatedPanel) iterator.next();
        personPanel.animate(); // update panel
    }

    repaint(); // paint all Components
}

private Iterator getPersonAnimatedPanelsIterator()
{

Fig. I.1 ElevatorView displays the elevator simulation model (part 8 of 18).
```java
// obtain iterator from List
synchronized (personAnimatedPanels)
{
    return new ArrayList(personAnimatedPanels).iterator();
}

// stop sound clip of Person walking
private void stopWalkingSound()
{
    // stop playing walking sound
    walkClip.stop();

    Iterator iterator = getPersonAnimatedPanelsIterator();

    // but if Person is still walking, then keep playing
    while (iterator.hasNext()) {
        AnimatedPanel panel = (AnimatedPanel) iterator.next();

        if (panel.getXVelocity() != 0)
            walkClip.loop();
    }
}

// returns Person AnimatedPanel with proper identifier
private AnimatedPanel getPersonPanel(PersonMoveEvent event)
{
    Iterator iterator = getPersonAnimatedPanelsIterator();

    while (iterator.hasNext()) {

        // get next AnimatedPanel
        AnimatedPanel personPanel =
            (AnimatedPanel) iterator.next();

        // return AnimatedPanel with identifier that matches
        if (personPanel.getID() == event.getID())
            return personPanel;
    }

    // return null if no match with correct identifier
    return null;
}

// invoked when Elevator has departed from Floor
public void elevatorDeparted(ElevatorMoveEvent moveEvent)
{
    String location =
        moveEvent.getLocation().getLocationName();

    // determine if Person is on Elevator
    Iterator iterator = getPersonAnimatedPanelsIterator();

    // Fig. I.1 ElevatorView displays the elevator simulation model (part 9 of 18).
```
while ( iterator.hasNext() ) {
    AnimatedPanel personPanel =
        ( AnimatedPanel ) iterator.next();
    double yPosition = personPanel.getPosition().getY();
    String panelLocation;

    // determine on which Floor the Person entered
    if ( yPosition > secondFloorPosition )
        panelLocation = FIRST_FLOOR_NAME;
    else
        panelLocation = SECOND_FLOOR_NAME;

    int xPosition =
        ( int ) personPanel.getPosition().getX();

    // if Person is inside Elevator
    if ( panelLocation.equals( location )
        && xPosition > PERSON_TO_BUTTON_DISTANCE + OFFSET ) {
        // remove Person AnimatedPanel from ElevatorView
        remove( personPanel );
        // add Person AnimatedPanel to Elevator
        elevatorPanel.add( personPanel, 1 );
        personPanel.setLocation( 2 * OFFSET, 9 * OFFSET );
        personPanel.setMoving( false );
        personPanel.setAnimating( false );
        personPanel.setVelocity( 0, 0 );
        personPanel.setCurrentFrame( 1 );
    }
} // end while loop

// determine Elevator velocity depending on Floor
if ( location.equals( FIRST_FLOOR_NAME ) )
    elevatorPanel.setVelocity( 0, -elevatorVelocity );
else
    if ( location.equals( SECOND_FLOOR_NAME ) )
        elevatorPanel.setVelocity( 0, elevatorVelocity );
// begin moving Elevator and play Elevator music
elevatorPanel.setMoving( true );
if ( elevatorClip != null )
    elevatorClip.play();
elevatorMusic.play();
} // end method elevatorDeparted

Fig. I.1 ElevatorView displays the elevator simulation model (part 10 of 18).
// invoked when Elevator has arrived at destination Floor
public void elevatorArrived( ElevatorMoveEvent moveEvent )
{
    // stop Elevator and music
    elevatorPanel.setMoving( false );
elevatorMusic.getSequencer().stop();

double xPos = elevatorPanel.getPosition().getX();
double yPos = 0;

    // set Elevator's position to either first or second Floor
if ( elevatorPanel.getYVelocity() < 0 )
yPos = secondFloorPosition - elevatorPanel.getHeight();
else
    yPos = firstFloorPosition - elevatorPanel.getHeight();
elevatorPanel.setPosition( xPos, yPos );
} // end method elevatorArrived

// invoked when Person has been created in model
public void personCreated( PersonMoveEvent personEvent )
{
    int personID = personEvent.getID();

    String floorLocation =
        personEvent.getLocation().getLocationName();

    // create AnimatedPanel representing Person
    AnimatedPanel personPanel =
        new AnimatedPanel( personID, personFrames );

    // determine where Person should be drawn initially
    // negative xPosition ensures Person drawn offscreen
double xPos = personPanel.getWidth();
double yPos = 0;

    if ( floorLocation.equals( FIRST_FLOOR_NAME ) )
yPos = firstFloorPosition +
        ( firstFloorPanel.getHeight() / 2 );
else
    if ( floorLocation.equals( SECOND_FLOOR_NAME ) )
yPos = secondFloorPosition +
        ( secondFloorPanel.getHeight() / 2 );
yPos -= personPanel.getHeight();
personPanel.setPosition( xPos, yPos );

Fig. I.1 **ElevatorView** displays the elevator simulation model (part 11 of 18).
// add some animations for each Person
int walkFrameOrder[] = { 1, 0, 1, 2 };
int pressButtonFrameOrder[] = { 1, 3, 3, 4, 4, 1 };
int walkAwayFrameOrder[] = { 6, 5, 6, 7 };
personPanel.addFrameSequence( walkFrameOrder );
personPanel.addFrameSequence( pressButtonFrameOrder );
personPanel.addFrameSequence( walkAwayFrameOrder );

// have Person begin walking to Elevator
personPanel.playAnimation( 0 );
personPanel.setLoop( true );
personPanel.setAnimating( true );
personPanel.setMoving( true );

// determine Person velocity
double time =
    ( double ) ( TIME_TO_BUTTON / ANIMATION_DELAY );
double xDistance = PERSON_TO_BUTTON_DISTANCE -
    2 * OFFSET + personPanel.getSize().width;
double xVelocity = xDistance / time;
personPanel.setVelocity( xVelocity, 0 );
personPanel.setAnimationRate( 1 );
walkClip.loop(); // play sound clip of Person walking

// store in personAnimatedPanels
synchronized( personAnimatedPanels )
{
    personAnimatedPanels.add( personPanel );
}
add( personPanel, 0 );

} // end method personCreated

// invoked when Person has arrived at Elevator
public void personArrived( PersonMoveEvent personEvent )
{
    // find Panel associated with Person that issued event
    AnimatedPanel panel = getPersonPanel( personEvent );

    if ( panel != null ) { // if Person exists
        // Person stops at Floor Button
        panel.setMoving( false );
        panel.setAnimating( false );
        panel.setCurrentFrame( 1 );
        stopWalkingSound();

        double xPosition = PERSON_TO_BUTTON_DISTANCE -
            ( panel.getSize().width / 2 );

        panel.setAnimationRate( 1 );
    }
}
double yPosition = panel.getPosition().getY();
panel.setPosition( xPosition, yPosition );
}
} // end method personArrived

// invoked when Person has pressed Button
public void personPressedButton( PersonMoveEvent personEvent )
{
    // find Panel associated with Person that issued event
    AnimatedPanel panel = getPersonPanel( personEvent );

    if ( panel != null ) { // if Person exists
        // Person stops walking and presses Button
        panel.setLoop( false );
        panel.playAnimation( 1 );

        panel.setVelocity( 0, 0 );
        panel.setMoving( false );
        panel.setAnimating( true );
        stopWalkingSound();
    }
} // end method personPressedButton

// invoked when Person has started to enter Elevator
public void personEntered( PersonMoveEvent personEvent )
{
    // find Panel associated with Person that issued event
    AnimatedPanel panel = getPersonPanel( personEvent );

    if ( panel != null ) {
        // determine velocity
double time = TIME_TO_ELEVATOR / ANIMATION_DELAY;

double distance =
elevatorPanel.getPosition().getX() -
panel.getPosition().getX() + 2 * OFFSET;

panel.setVelocity( distance / time, -1.5 );

        // Person starts walking
        panel.setMoving( true );
        panel.playAnimation( 0 );
        panel.setLoop( true );
    }
} // end method personEntered

// invoked when Person has departed from Elevator
public void personDeparted( PersonMoveEvent personEvent)
{

Fig. I.1 ElevatorView displays the elevator simulation model (part 13 of 18).
Appendix I Elevator View (on CD) 1451

652 // find Panel associated with Person that issued event
653 AnimatedPanel panel = getPersonPanel( personEvent );
654
655 if ( panel != null ) { // if Person exists
656
657 // determine velocity (in opposite direction)
658 double time = TIME_TO_BUTTON / ANIMATION_DELAY;
659 double xVelocity = - PERSON_TO_BUTTON_DISTANCE / time;
660
661 panel.setVelocity( xVelocity, 0 );
662
663 // remove Person from Elevator
664 elevatorPanel.remove( panel );
665
666 double xPosition =
667 PERSON_TO_ELEVATOR_DISTANCE + 3 * OFFSET;
668 double yPosition = 0;
669
670 String floorLocation =
671 personEvent.getLocation().getLocationName();
672
673 // determine Floor onto which Person exits
674 if ( floorLocation.equals( FIRST_FLOOR_NAME ) )
675 yPosition = firstFloorPosition +
676 ( firstFloorPanel.getHeight() / 2 );
677 else
678
679 if ( floorLocation.equals( SECOND_FLOOR_NAME ) )
680 yPosition = secondFloorPosition +
681 ( secondFloorPanel.getHeight() / 2 );
682
683 yPosition -= panel.getHeight();
684
685 panel.setPosition( xPosition, yPosition );
686
687 // add Person to ElevatorView
688 add( panel, 0 );
689
690 // Person starts walking
691 panel.setMoving( true );
692 panel.setAnimating( true );
693 panel.playAnimation( 2 );
694 panel.setLoop( true );
695 walkClip.loop();
696 }
697 } // end method PersonDeparted
698
699 // invoked when Person has exited simulation
700 public void personExited( PersonMoveEvent personEvent)
701 {
702 // find Panel associated with Person that issued moveEvent
703 AnimatedPanel panel = getPersonPanel( personEvent );
704
Fig. I.1 ElevatorView displays the elevator simulation model (part 14 of 18).
if ( panel != null ) { // if Person exists
    panel.setMoving( false );
    panel.setAnimating( false );

    // remove Person permanently and stop walking sound
    synchronized( personAnimatedPanels )
    {
        personAnimatedPanels.remove( panel );
    }
    remove( panel );
    stopWalkingSound();
}
} // end method personExited

// invoked when Door has opened in model
public void doorOpened( DoorEvent doorEvent )
{
    // get DoorEvent Location
    String location =
        doorEvent.getLocation().getLocationName();

    // play animation of Door opening
    doorPanel.playAnimation( 0 );
    doorPanel.setAnimationRate( 2 );
    doorPanel.setDisplayLastFrame( true );

    // play sound clip of Door opening
    if ( doorOpenClip != null )
        doorOpenClip.play();

} // end method doorOpened

// invoked when Door has closed in model
public void doorClosed( DoorEvent doorEvent )
{
    // get DoorEvent Location
    String location =
        doorEvent.getLocation().getLocationName();

    // play animation of Door closing
    doorPanel.playAnimation( 1 );
    doorPanel.setAnimationRate( 2 );
    doorPanel.setDisplayLastFrame( true );

    // play sound clip of Door closing
    if ( doorCloseClip != null )
        doorCloseClip.play();

} // end method doorClosed

Fig. I.1 ElevatorView displays the elevator simulation model (part 15 of 18).
public void buttonPressed( ButtonEvent buttonEvent ) {

   // get ButtonEvent Location
   String location =
      buttonEvent.getLocation().getLocationName();

   // press Elevator Button if from Elevator
   if ( location.equals( ELEVATOR_NAME ) ) {
      elevatorButtonPanel.playAnimation( 0 );
      elevatorButtonPanel.setDisplayLastFrame( true );
   }

   // press Floor Button if from Floor
   else

      if ( location.equals( FIRST_FLOOR_NAME ) ) {
         firstFloorButtonPanel.playAnimation( 0 );
         firstFloorButtonPanel.setDisplayLastFrame( true );
      }

      else

         if ( location.equals( SECOND_FLOOR_NAME ) ) {
            secondFloorButtonPanel.playAnimation( 0 );
            secondFloorButtonPanel.setDisplayLastFrame( true );
         }

         else

            if ( buttonClip != null )
               buttonClip.play(); // play button press sound clip

   } // end method buttonPressed

// invoked when Button has been reset in model
public void buttonReset( ButtonEvent buttonEvent ) {

   // get ButtonEvent Location
   String location =
      buttonEvent.getLocation().getLocationName();

   // reset Elevator Button if from Elevator
   if ( location.equals( ELEVATOR_NAME ) ) {
      // return to first frame if still animating
      if ( elevatorButtonPanel.isAnimating() )
         elevatorButtonPanel.setDisplayLastFrame( false );
      else
         elevatorButtonPanel.setCurrentFrame( 0 );
   }

   // reset Floor Button if from Floor
   else

      if ( buttonClip != null )
         buttonClip.play(); // play button press sound clip

} // end method buttonReset

Fig. I.1 ElevatorView displays the elevator simulation model (part 16 of 18).
if ( location.equals( FIRST_FLOOR_NAME ) ) {
    // return to first frame if still animating
    if ( firstFloorButtonPanel.isAnimating() )
        firstFloorButtonPanelsetDisplayLastFrame( false );
    else
        firstFloorButtonPanel setCurrentFrame( 0 );
}
else

if ( location.equals( SECOND_FLOOR_NAME ) ) {
    // return to first frame if still animating
    if ( secondFloorButtonPanel.isAnimating() )
        secondFloorButtonPanel.setDisplayLastFrame( false );
    else
        secondFloorButtonPanel.setCurrentFrame( 0 );
}
} // end method buttonReset

// invoked when Bell has rung in model
public void bellRang( BellEvent bellEvent )
{
    bellPanel.playAnimation( 0 ); // animate Bell
    if ( bellClip != null ) // play Bell sound clip
        bellClip.play();
}

// invoked when Light turned on in model
public void lightTurnedOn( LightEvent lightEvent )
{
    // turn on Light in Elevator
    elevatorLightPanel.setCurrentFrame( 1 );
    String location =
        lightEvent.getLocation().getLocationName();
    // turn on Light on either first or second Floor
    if ( location.equals( FIRST_FLOOR_NAME ) )
        firstFloorLightPanel.setCurrentFrame( 1 );
    else
        if ( location.equals( SECOND_FLOOR_NAME ) )
            secondFloorLightPanel.setCurrentFrame( 1 );
} // end method lightTurnedOn

Fig. I.1 ElevatorView displays the elevator simulation model (part 17 of 18).
I.2 Class Objects

The ElevatorView is a JPanel with a series of other JPanel “children” added to it. Each JPanel provides a visual representation of an object from the model. For example, the ElevatorView contains ImagePanels, MovingPanels and AnimatedPanels to represent the Elevator, Persons, the ElevatorShaft, the Buttons on the Floors, the Button in the Elevator, the Doors on the Floors, the Door in the Elevator, the Lights on the Floors, the two Floors and the Bell. Figure I.2 lists the ElevatorView’s objects and their counterparts in the model.

```java
// invoked when Light turned off in model
public void lightTurnedOff( LightEvent lightEvent )
{
    // turn off Light in Elevator
    elevatorLightPanel.setCurrentFrame( 0 );

    String location =
        lightEvent.getLocation().getLocationName();

    // turn off Light on either first or second Floor
    if ( location.equals( FIRST_FLOOR_NAME ) )
        firstFloorLightPanel.setCurrentFrame( 0 );
    else
        if ( location.equals( SECOND_FLOOR_NAME ) )
            secondFloorLightPanel.setCurrentFrame( 0 );

} // end method lightTurnedOff

// return preferred size of ElevatorView
public Dimension getPreferredSize()
{
    return new Dimension( VIEW_WIDTH, VIEW_HEIGHT );
}

// return minimum size of ElevatorView
public Dimension getMinimumSize()
{
    return getPreferredSize();
}

// return maximum size of ElevatorView
public Dimension getMaximumSize()
{
    return getPreferredSize();
}
```

Fig. I.1 ElevatorView displays the elevator simulation model (part 18 of 18).
Lines 108–128 of class ElevatorView declare the objects in the second column of Fig. I.2. The firstFloorPanel (line 108), secondFloorPanel (line 109) and elevatorShaftPanel (line 110) are ImagePanels, because neither the Floors nor the ElevatorShaft move in the simulation. The elevatorPanel (line 115) is a MovingPanel, because the Elevator's only function is to move between Floors. The firstFloorButtonPanel (line 118), secondFloorButtonPanel (line 119) and elevatorButtonPanel (line 120) are AnimatedPanels, because each object animates when the associated Button in the model is pressed or reset. The bellPanel (line 121) is an AnimatedPanel to animate the ringing of the Bell. The firstFloorLightPanel (line 123) and secondFloorLightPanel (line 124) are AnimatedPanels, because these objects animate when the associated Light turns on or off. The doorPanel (line 125) is an AnimatedPanel to animate the opening and closing of the Door. Note that the ElevatorView shows only the Door in the Elevator. The ElevatorView does not show the Doors on the Floors, which enables us to show the Elevator's interior (these Doors would obstruct the objects inside the Elevator). Lastly, the personAnimatedPanels (line 128) is a List of AnimatedPanels, because there can exist several Person objects during execution—the ElevatorView must need to store dynamically the AnimatedPanels associated with Persons in the model.

We add to the ElevatorView three more elements that we assume to be parts of the Elevator (Fig. I.3), although the model does not represent these elements—a light inside the Elevator of type AnimatedPanel called elevatorLightPanel (line 122), a ceiling over the Elevator of type ImagePanel called ceilingPanel (line 112), and wallpaper inside the building of type ImagePanel called wallPanel (line 111).
In addition, the class diagram of Fig. 22.9 shows that the ElevatorView contains one instance each of classes SoundEffects and ElevatorMusic. The SoundEffects object generates the AudioClips used to play sound effects, such as the Door opening and a Person walking. Lines 131–136 declare all AudioClips, line 139 declares the ElevatorMusic object, and line 328 (in method initializeAudio, which we discuss later in this section) declares the SoundEffects object.

### I.3 Class Constants

The ElevatorView uses constants to specify or obtain such information as

- The initial placement of objects in the ElevatorView
- The rate at which the ElevatorView redraws the screen (animation rate)
- The names of image files used by the ImagePanels
- The names of sound files used by the SoundEffects object and the ElevatorMusic
- The distances in pixels the ImagePanels representing the Elevator and Person must travel
- The times needed to travel these distances

Lines 23–24 declare int constants VIEW_WIDTH and VIEW_HEIGHT, which specify the ElevatorView’s dimensions. Method getPreferredSize (lines 881–884) returns this dimension. Method pack of class ElevatorSimulation uses this method to obtain the ElevatorView’s dimension to place the ElevatorView in the GUI properly.

The ElevatorView has a null layout, so we may place ImagePanels in any x-y coordinate in the ElevatorView. Line 27 of class ElevatorView declares int constant OFFSET, which helps to determine the exact positions of objects in the ElevatorView. Line 30 declares int constant ANIMATION_DELAY, which specifies the number of milliseconds between animation frames. In our simulation, we initialize ANIMATION_DELAY to 50 milliseconds. Lines 46–95 declare String constants specifying the image files used to instantiate the ImagePanels. Lines 98–105 declare the String constant specifying the audio files used to instantiate the AudioClips and the ElevatorMusic.

Line 33 declares the int constant PERSON_TO_BUTTON_DISTANCE, which represents the horizontal distance between the on-screen location of the firstFloorBut-
t onPanel or secondFloorButtonPanel and the initial on-screen location of an AnimatedPanel associated with Person. This AnimatedPanel uses this constant to calculate the travel distance to the firstFloorButtonPanel or secondFloorButtonPanel. The firstFloorButtonPanel and secondFloorButtonPanel use the constant to position themselves on screen. Line 34 declares the int constant BUTTON_TO_ELEVATOR_DISTANCE, describing the horizontal distance between the firstFloorButtonPanel or secondFloorButtonPanel and the elevatorPanel. The AnimatedPanel associated with a Person uses this constant to determine the travel distance when entering the elevatorPanel.

Line 39 declares int constant TIME_TO_BUTTON, which represents this AnimatedPanel’s travel time to the firstFloorButtonPanel or secondFloorButtonPanel. Line 40 declares int constant TIME_TO_ELEVATOR, which represents the time the AnimatedPanel associated with a Person needs to enter the elevatorPanel from the firstFloorButtonPanel or secondFloorButtonPanel. Using the equation rate = distance / time, the AnimatedPanel associated with the Person can determine the velocity needed to travel. Similarly, line 43 declares int constant ELEVATOR_TRAVEL_TIME, which represents the elevatorPanel’s travel time between the firstFloorPanel and secondFloorPanel—lines 166–169 use this constant to determine double attribute elevatorVelocity (line 149).

I.4 Class constructor

The responsibilities of the ElevatorView constructor (lines 152–174) are

- To instantiate all ImagePanel
- To add all ImagePanel to the ElevatorView
- To initialize the audio objects
- To compute the elevatorPanel’s initial velocity and distance traveled
- To start the animation Timer

Lines 157 calls private method instantiatePanels (lines 177–299), which instantiates all ImagePanel in the ElevatorView. Lines 180–181 instantiate the firstFloorPanel and secondFloorPanel, and lines 184–190 set these objects’ positions—the ElevatorView positions the firstFloorPanel on the bottom of the screen and positions the secondFloorPanel in the vertical center of the screen. Line 192 instantiates the wallPanel ImagePanel. The ElevatorView does not need to calculate the position for the wallPanel, because the wallPanel’s default screen position (i.e., xPosition = 0, yPosition = 0) is correct. Lines 195–202 and lines 205–210 instantiate and position the elevatorShaftPanel and ceilingPanel ImagePanels, respectively. The ElevatorView positions the elevatorShaftPanel in the right of the screen and positions the ceilingPanel above the elevatorShaftPanel. Lines 213–217 instantiate the elevatorPanel and position it over the elevatorShaftPanel above the firstFloorPanel. Lines 220–229 instantiate the firstFloorButtonPanelAnimatedPanel, place it next to the elevatorShaftPanel, then create a frame sequence Button pressed animation. Lines 232–240 perform the same actions on the secondFloorButtonPanel. Lines 243–249 instantiate the firstFloorLightPanelAnimatedPanel and place it to
the left of the elevatorShaftPanel but above the firstFloorButtonPanel. Lines 251–256 position the secondFloorLightPanel above the secondFloorButtonPanel. Lines 259–269 instantiate the doorPanelAnimatedPanel, place it relative to the elevatorPanel’s position (because the elevatorPanel will contain the doorPanel) and assign frame sequences describing the Door animation opening and closing. Lines 272–273 instantiate the elevatorLightPanelAnimatedPanel and position it over the elevatorPanel and to the left of the doorPanel. Lines 276–283 instantiate the bellPanel, position it below the elevatorLightPanel, then assign a frame sequence describing the Bell ringing animation. Lines 286–294 instantiate the elevatorButtonPanel, position it in the center of the elevatorPanel and assign a frame sequence describing the Button pressed animation. Lastly, line 297 instantiates the ArrayList holding the AnimatedPanel objects associated with the Persons in the model.

After the ElevatorView constructor has called method instantiatePanels, the constructor calls method placePanelsOnView (lines 302–322), which adds all instantiated Panels to the ElevatorView. Lines 317–320 add the doorPanel, elevatorLightPanel, bellPanel and elevatorButtonPanel to the elevatorPanel. The ElevatorView constructor then calls method initializeAudio (lines 325–342). Lines 328–329 instantiate a SoundEffects object, and lines 331–336 use method getAudioClip of the SoundEffects object to return the AudioClips for the simulation. Method play of class AudioClip plays the AudioClip—the ElevatorView uses this method for sounds that do not repeat, such as the Bell ring. Method loop of class AudioClip plays the clip continually—the ElevatorView uses this method for sounds that repeat, such as the sound of footsteps. Lines 339–340 instantiate the ElevatorMusic object and ensure that the MIDI data is valid.

Finally, lines 162–163 (in the ElevatorView constructor) calculate the distance between the two Floors (i.e., the distance the elevatorPanel will travel). Lines 166–169 use the equation rate = distance / time to determine the elevatorPanel’s velocity when traveling. Finally, line 172 calls method startAnimation, which starts the animation timer.


We present an object diagram that lists all objects in the ElevatorView. Recall that an object diagram provides a snapshot of the structure when the system is running. The object diagram of Fig. I.4 represents the ElevatorView after invoking the constructor.
The ElevatorView object links (contains an association) with all objects presented in Fig. I.4. The elevatorPanel links with objects elevatorLightPanel, bellPanel, doorPanel and elevatorButtonPanel. This association provides a visualization of what is happening in the model—the Elevator contains a Light, Bell, Door and Button. The SoundEffects object links with the AudioClip objects, because the SoundEffects object generates the AudioClip objects.

### I.5 Event Handling

Figure 13.19 specified that the ElevatorView implements interface ElevatorModelListener, which implements all interfaces in the simulation. The ElevatorSimulation registers the ElevatorView as a listener for events from the ElevatorModel; in other words, the ElevatorModel sends all events generated in the model to the ElevatorView.

Every method implementing an interface receives an event object of type ElevatorModelEvent (or a subclass) as a parameter. For example, the doorOpened method
receives a `DoorEvent`. Appendix G contains further reference on events and listeners. The following sections discuss the types of events that the `ElevatorView` handles.

### I.5.1 ElevatorMoveEvent types

The `ElevatorModel` sends an `ElevatorMoveEvent` when the `Elevator` has either departed or arrived in the model. The `ElevatorModel` invokes method `elevatorDeparted` (lines 435–493) when the `Elevator` has departed from a `Floor`. Lines 441–475 determine if an `AnimatedPanel` associated with a `Person` overlaps the `elevatorPanel` by iterating `personAnimatedPanels` and testing whether any AnimatedPanel in the List has an on-screen x-coordinate greater than that of the `elevatorPanel`. If this is the case, then the `Person` is inside the `Elevator`, and lines 465–468 add the `AnimatedPanel` associated with that `Person` to the `elevatorPanel`. Regardless of whether a `Person` is inside the `Elevator`, lines 478–515 set the `elevatorPanel`’s velocity according to the direction the `Elevator` must travel. Line 500 plays the `elevatorMusic`.

The `ElevatorModel` invokes method `elevatorArrived` (lines 496–515) when the `Elevator` has arrived at a `Floor`. Line 499 stops the `elevatorPanel`, and line 512 stops the `elevatorMusic`. Lines 506–513 change the direction of the `elevatorPanel` for the next travel.

### I.5.2 PersonMoveEvent types

The `ElevatorModel` sends a `PersonMoveEvent` when a `Person` has performed some action in the model that the `ElevatorView` must represent. The `ElevatorModel` invokes method `personCreated` (lines 518–582) when the model instantiates a new `Person`. Lines 526–527 instantiate an `AnimatedPanel` for a `Person`. Lines 531–545 determine on which `Floor` to situate the `AnimatedPanel`, depending on the `Floor` on which the event was generated. Lines 548–553 add frame sequences to the `AnimatedPanel` describing the `Person` walking and pressing a `Button`. Lines 556–572 animate the `Person` walking, determine the `Person`’s velocity necessary to reach the `Button` on the `Floor` and play the sound effect of footsteps. Lastly, lines 575–580 add the `AnimatedPanel` associated with the `Person` to List `personAnimatedPanels`, using a synchronized block (lines 575–578) to guarantee no other object can access the List.

The `ElevatorModel` invokes method `personArrived` (lines 585–604) when a `Person` has arrived at the `Elevator`. Line 588 calls method `getPersonPanel` (lines 414–432), which determines the `AnimatedPanel` associated with the `Person` that issued the event. Specifically, method `getPersonPanel` iterates List `personAnimatedPanels` and returns the `AnimatedPanel` whose identifier matches the identifier of the `PersonMoveEvent`. Lines 590–603 in method `personArrived` stop this `AnimatedPanel` from moving. Line 596 stops the sound of footsteps by calling method `stopWalkingSound` (lines 397–411), which stops the `AudioClip` playing the footstep sound only if no `Persons` are walking.

The `ElevatorModel` invokes method `personPressedButton` (lines 607–623) when a `Person` pressed a `Button`. Line 610 determines the `AnimatedPanel` associated with the `Person` who pressed the `Button`. Line 616 calls method `playAnimation`, which plays the animation sequence of that `Person` pressing the `Button`. 
The ElevatorModel invokes method personEntered (lines 626–647) when a Person is about to enter the Elevator. Line 629 retrieves the AnimatedPanel associated with the Person entering the Elevator. Line 634–640 determine the velocity needed to walk into the Elevator. Line 643–645 animate this AnimatedPanel to walk in the elevatorPanel.

The ElevatorModel invokes method personDeparted (lines 650–697) when a Person is about to exit the Elevator. Line 653 determines the AnimatedPanel associated with the Person departing from the Elevator. Lines 658–661 determine that Person’s velocity needed to walk to across the Floor to exit the simulation. Lines 664–688 position the AnimatedPanel associated with the Person on the Floor in front of the Elevator by removing the AnimatedPanel from the elevatorPanel and adding the AnimatedPanel to the ElevatorView. Lines 691–695 animate this AnimatedPanel to walk across either the firstFloorPanel or secondFloorPanel and start the sound of footsteps.

The ElevatorModel invokes method personExited (lines 700–718) when a Person has exited from the simulation. Line 703 determines the AnimatedPanel associated with the Person who exited the simulation. Lines 711–716 remove the AnimatedPanel associated with that Person from the ElevatorView and stop the sound of footsteps.

1.5.3 DoorEvent types

The ElevatorModel sends a DoorEvent to the ElevatorView when a Door has opened or closed in the model. The ElevatorModel invokes method doorOpened (lines 721–736) when a Door has opened. Lines 724–730 animate the doorPanel opening, and lines 733–734 plays the doorOpenClip, which is the sound effect associated with the Door’s opening.

The ElevatorModel invokes method doorClosed (lines 739–754) when a Door has closed. Lines 742–748 animate the doorPanel closing, and lines 751–752 plays the doorClosedClip, which is the sound effect associated with the Door’s closing.

1.5.4 ButtonEvent types

The ElevatorModel sends a ButtonEvent to the ElevatorView when a Button has been pressed or reset in the model. The ElevatorModel invokes method buttonPressed (lines 757–786) when a Button has been pressed. Lines 760–761 determine the Location where the Button was pressed. If the Location is the Elevator, then lines 764–767 play the Button pressed animation inside the Elevator. If the Location is the first Floor, then lines 772–767 play the Button pressed animation on the first Floor. If the Location is the second Floor, then lines 778–781 play the Button pressed animation on the second Floor being pressed. Lines 783–784 play the buttonClip, which is the sound effect associated with the Button being pressed.

The ElevatorModel invokes method buttonReset (lines 789–829) when a Button has been reset. Lines 792–793 determine the Location where the Button was reset. If the Location is the Elevator, then lines 796–803 change the elevatorButtonPanel’s image to that of the Button reset. If the Location is the first Floor, then lines 808–816 change the firstFloorButtonPanel’s image associated with the
Appendix I Elevator View (on CD) 1463

Button reset. If the Location is the second Floor, then lines 819–827 change the secondFloorButtonPanel’s image associated with the Button reset.

1.5.5 BellEvent types

The ElevatorModel sends a BellEvent to the ElevatorView by invoking method bellRang (lines 832–838) when a Bell has rung in the model. Line 834 animates the bellPanel, and lines 836–837 play the bellClip, which is the sound effect associated with the Bell ringing.

1.5.6 LightEvent types

The ElevatorModel sends a LightEvent to the ElevatorView when a Light has changed state in the model. The ElevatorModel invokes method light-TurnedOn (lines 841–858) when a Light has turned on. Line 844 turns on elevator-LightPanel. Lines 846–856 determine on which Floor the Light has turned on, then illuminates the AnimatedPanel associated with that Light in the ElevatorView.

The ElevatorModel invokes method lightTurnedOff (lines 861–878) when a Light has turned off. Line 864 turns on elevatorLightPanel. Lines 866–876 determine on which Floor the Light has turned on, then turns off the AnimatedPanel associated with that Light in the ElevatorView.

1.6 Component Diagrams Revisited

In Section 13.17, we introduced the component diagram for the elevator simulation, and in Appendix G and Appendix H, we added components to packages event and model, respectively. Figure I.5 presents the component diagram for package view, which contains components ElevatorView.java, ImagePanel.java, MovingPanel.java, AnimatedPanel.java, ElevatorMusic.java and SoundEffects.java. ElevatorView.java aggregates packages images, sounds and event. Packages images and sounds contain all image files and sound files (components) used by ElevatorView.java, respectively. The diagram does not show the components of these directories, because there exist far too many graphics and audio files to represent on one page—the contents of these packages can be found in the directory structures

    com/deitel/jhtp4/elevator/view/images
    com/deitel/jhtp4/elevator/view/sounds

(i.e., in the images and sounds directory where the classes for the view are located in the file system).

1.7 Conclusion

Congratulations! You have completed an “industrial-strength” OOD/UML case study. You are well prepared to tackle more substantial design problems and to go on to deeper study of OOD with the UML. Hopefully you have developed a greater appreciation and understanding of design and implementation processes. Now, you can use Java to implement substantial object-oriented system designs generated by the UML. We hope you have
enjoyed using Java and the UML to construct this case study while learning what features
the two technologies have to offer. In addition, we hope you have enjoyed using Java’s
GUI, graphics and sound capabilities, while learning important object-oriented and Java-
related concepts, such as classes, objects, GUI construction, inheritance, event handling
and multithreading.

Fig. I.5  Component diagram for package view.
Career Opportunities (on CD)

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
Outline

J.1 Introduction
J.2 Resources for the Job Seeker
J.3 Online Opportunities for Employers
  J.3.1 Posting Jobs Online
  J.3.2 Problems with Recruiting on the Web
  J.3.3 Diversity in the Workplace
J.4 Recruiting Services
  J.4.1 Testing Potential Employees Online
J.5 Career Sites
  J.5.1 Comprehensive Career Sites
  J.5.2 Technical Positions
  J.5.3 Wireless Positions
  J.5.4 Contracting Online
  J.5.5 Executive Positions
  J.5.6 Students and Young Professionals
  J.5.7 Other Online Career Services
J.6 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises • Works Cited

J.1 Introduction

There are approximately 40,000 career-advancement services on the Internet today. 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.

### J.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.

**J.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

J.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.web-hire.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.
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

---

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*. 
J.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.

J.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
Appendix J  Career Opportunities (on CD)  1473

according to demographics (African American, Hispanic, alternative lifestyle, etc.) or by subject (employer, position, etc.) via hundreds of links. Featured sites include Bilin-
gualJobs.com, Latin World and American Society for Female Entrepreneurs.

Many sites have sections dedicated to job seekers with disabilities. In addition to pro-
viding 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.

J.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 in-
terviewing qualified candidates and making the best matches possible.

Advantage Hiring, Inc. (www.advantagehiring.com) provides employers with
a resume-screening service. When a prospective employee submits a resume for a partic-
ular position, Advantage Hiring, Inc. presents Net-Interview™, a small questionnaire to
supplement the information presented on the resume. The site also offers SiteBuilder, a ser-
vice that helps employers build an employee recruitment site. An online demonstration can
be found at 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. 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.recruit-
soft.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 Peo-
pleCheck™. 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 arti-
cles 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 thou-
sands 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™.
J.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.advisor.sorteam.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.
J.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.

J.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.
J.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 J.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](http://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.

J.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 a interview and discussion. Jobs4wireless.com also provides job seekers with employment opportunities in the wireless industry.

J.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 J.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.

J.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.

J.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-
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), [brassring-campus.com](http://brassring-campus.com) and [collegegrads.com](http://collegegrads.com).

### J.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,** organized by geographic location, helps job seekers find opportunities that allow them to bring their pets to work, and **cooljobs.com** is a searchable database of unique job opportunities.
J.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 searchers 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.opportunitynoc.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.
Students seeking internships can search job listings on this site. It also features City Intern, to help interns become acquainted with a new location.

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
demographic open-ended question
end-to-end recruiting solutions pay-per-hire
entry-level position request for proposal (RFP)
online contracting service resume-filtering software

**SELF-REVIEW EXERCISES**

**J.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.

**J.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

J.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.

J.2  a) 40,000. b) diversity. c) 5 million. d) resume-filtering software. e) eligible.

EXERCISES

J.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.

J.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 ________.

J.5  Define the following
   a) corporate culture
   b) pay-per-hire
   c) request for proposal (RFP)
   d) resume-filtering software

J.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?

J.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.
K.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
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.

K.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.

K.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 ALPHA</td>
<td>0xCD 0x91</td>
<td>0x0391</td>
<td>0x00000391</td>
</tr>
<tr>
<td>CJK UNIFIED IDEOGRAPH-4E95</td>
<td>0xE4 0xBA 0x95</td>
<td>0x4E95</td>
<td>0x00004E95</td>
</tr>
<tr>
<td>OLD ITALIC LETTER A</td>
<td>0xF0 0x80 0x83 0x80</td>
<td>0xDC00 0xDF00</td>
<td>0x00010300</td>
</tr>
</tbody>
</table>

Fig. K.1 Correlation between the three encoding forms.

Fig. K.2 Various glyphs of the character A.
K.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.

K.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.

### K.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.]

```java
// Fig. K.3: Unicode.java
// Demonstrating how to use Unicode in Java programs.

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

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

**Fig. K.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\u0020\u0074\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\u0020Unicode\u0021\" );
        cyrillic.setToolTipText( "This is Russian" );
        container.add( cyrillic );

        french = new JLabel( \"\u0042\u0069\u0065\u006E\u0076\u006F\u0020Unicode\u0021\" );
        french.setToolTipText( "This is French" );
        container.add( french );

        german = new JLabel( \"\u0057\u0069\u006C\u006B\u006F\u006D\u0020\u007A\u0020Unicode\u0021\" );
        german.setToolTipText( "This is German" );
        container.add( german );

        japanese = new JLabel( \"Unicode\u3078\u3087\u3045\u305D\u0021\" );
        japanese.setToolTipText( "This is Japanese" );
        container.add( hiragana );

        portuguese = new JLabel( \"\u0053\u00E9\u006A\u0020\u0042\u0065\u006D\u0076\u0069\u006E\u0064\u0020Unicode\u0021\" );
        portuguese.setToolTipText( "This is Portuguese" );
        container.add( portuguese );
    }
}

Fig. K.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 `\u`yyyy, 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.zh` 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`.]

Fig. K.3 Java program that uses Unicode encoding (part 3 of 3).
K.7 Character Ranges

The Unicode Standard assigns code values, which range from \( \text{0000 (Basic Latin)} \) to \( \text{E007F (Tags)} \), 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 \uyyyy 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**

\uyyyy escape sequence, portability
ASCII, script
block, surrogate
code value, Unicode Consortium
diacritic, Unicode design basis
double-byte character set (DBCS), Unicode Standard
efficient (Unicode design basis), Unicode Transformation Format (UTF)
encode, uniform (Unicode design basis)
escape sequence, universal (Unicode design basis)
glyph, UTF-8
hexadecimal notation, UTF-16
localization, UTF-32
multi-byte character set (MBCS)

**SELF-REVIEW EXERCISES**

K.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 _________.

K.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**

K.1 a) localize. b) encoding. c) universal, efficient, uniform, unambiguous. d) character. e) portable.
K.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

K.3  Navigate to the Unicode Consortium Web site (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 (‘).’

K.4  Describe the Unicode Standard design basis.

K.5  Define the following terms:
   a) code value.
   b) surrogates.
   c) Unicode Standard.

K.6  Define the following terms:
   a) UTF-8.
   b) UTF-16.
   c) UTF-32.

K.7  Describe a scenario where it is optimal to store your data in UTF-16 format.

K.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.